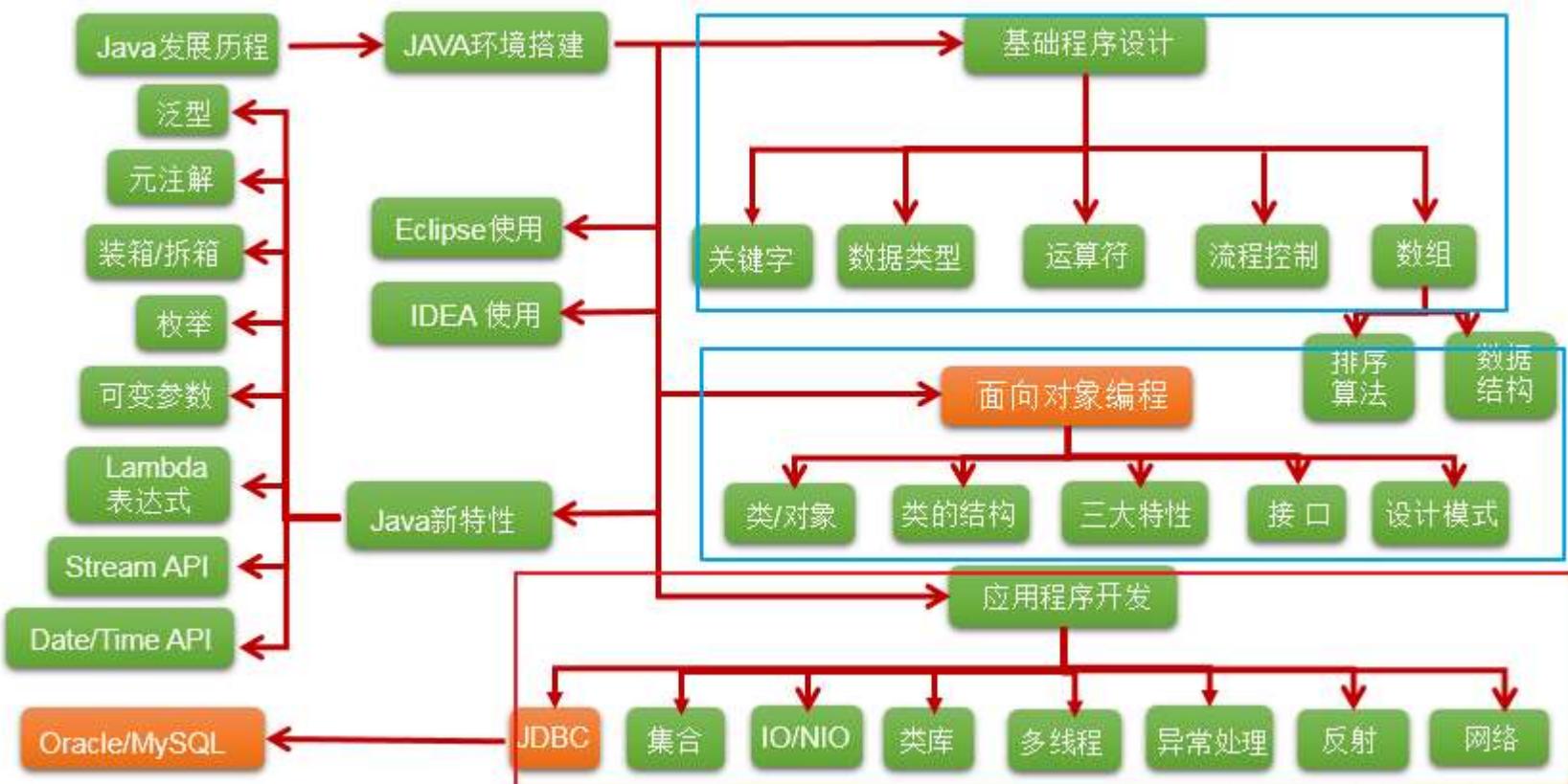


课程整体内容概述



第一部分：编程语言核心结构

主要知识点：变量、基本语法、分支、循环、数组、...

第二部分：Java面向对象的核心逻辑

主要知识点：OOP、封装、继承、多态、接口、...

第三部分：开发Java SE高级应用程序

主要知识点：异常、集合、I/O、多线程、反射机制、网络编程、.....

第四部分：实训项目

项目一：家庭收支记账软件

项目二：客户信息管理软件

项目三：开发团队人员调度软件

附加项目一：银行业务管理软件

附件项目二：单机考试管理软件

项目一：讲完流程控制时，可以做。第二章结束

项目二：讲完第四章面向对象（上），可以做

项目三：讲完第七章异常处理以后，可以做

附加项目一：讲完第七章异常处理以后，可以做

附加项目二：讲完第11章IO流以后，可以做

1. 基础常识

软件：即一系列按照特定顺序组织的计算机数据和指令的集合。分为：系统软件 和 应用软件

系统软件：windows , mac os , linux , unix, android, ios,....

应用软件：word , ppt, 画图板,...

人机交互方式： 图形化界面 vs 命令行方式

应用程序 = 算法 + 数据结构

常用DOS命令：

● 常用的DOS命令

- **dir**： 列出当前目录下的文件以及文件夹
- **md**： 创建目录
- **rd**： 删除目录
- **cd**： 进入指定目录
- **cd..**： 退回到上一级目录
- **cd**： 退回到根目录
- **del**： 删除文件
- **exit**： 退出 dos 命令行
- ✓ 补充： echo javase>1.doc

● 常用快捷键

- ← →： 移动光标
- ↑ ↓： 调阅历史操作命令
- Delete和Backspace： 删除字符

2. 计算机语言的发展迭代史

第一代：机器语言

第二代：汇编语言

第三代：高级语言

- > 面向过程： C,Pascal、Fortran
- > 面向对象： Java,JS,Python,Scala,...

3. Java语言版本迭代概述

- ⌚ 1991年 Green项目，开发语言最初命名为Oak (橡树)
- ⌚ 1994年，开发组意识到Oak 非常适合于互联网
- ⌚ 1996年，发布JDK 1.0，约8.3万个网页应用Java技术来制作
- ⌚ 1997年，发布JDK 1.1，JavaOne会议召开，创当时全球同类会议规模之最
- ⌚ 1998年，发布JDK 1.2，同年发布企业平台J2EE
- ⌚ 1999年，Java分成J2SE、J2EE和J2ME，JSP/Servlet技术诞生
- ⌚ 2004年，发布里程碑式版本： JDK 1.5，为突出此版本的重要性，更名为JDK 5.0
- ⌚ 2005年，J2SE -> JavaSE, J2EE -> JavaEE, J2ME -> JavaME
- ⌚ 2009年，Oracle公司收购SUN，交易价格74亿美元
- ⌚ 2011年，发布JDK 7.0
- ⌚ 2014年，发布JDK 8.0，是继JDK 5.0以来变化最大的版本
- ⌚ 2017年，发布JDK 9.0，最大限度实现模块化
- ⌚ 2018年3月，发布JDK 10.0，版本号也称为18.3

④ 2018年9月，发布JDK 11.0，版本号也称为18.9

4. Java语言应用的领域：

> Java Web开发：后台开发

> 大数据开发：

> Android应用程序开发：客户端开发

5. Java语言的特点

> 面向对象性：

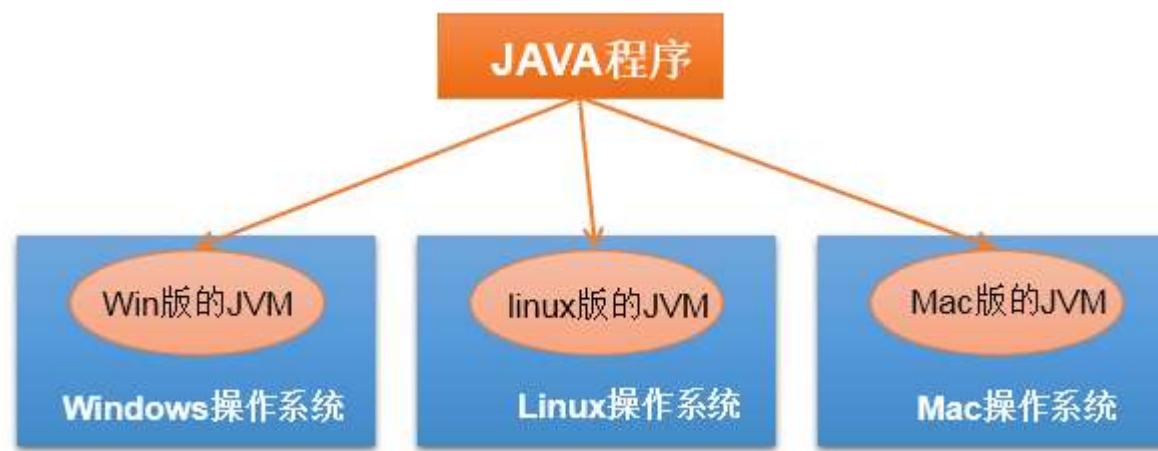
两个要素：类、对象

三个特征：封装、继承、多态

> 健壮性：① 去除了C语言中的指针 ② 自动的垃圾回收机制 --> 仍然会出现内存溢出、内存泄漏

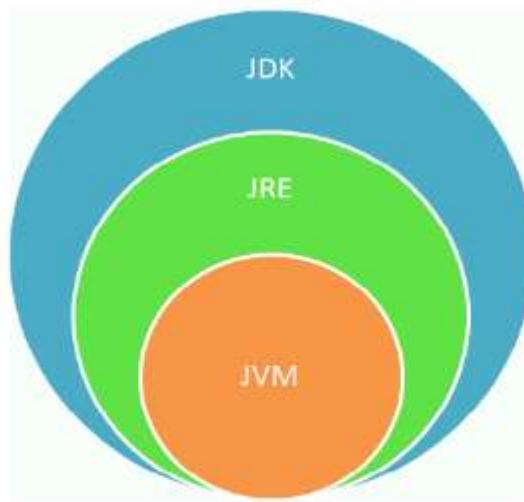
> 跨平台型：`write once, run anywhere`：一次编译，到处运行

功劳归功于：JVM



1. 开发环境的搭建（重点）

1.1 JDK、JRE、JVM的关系



- **JDK = JRE + 开发工具集**（例如**Javac**编译工具等）
- **JRE = JVM + Java SE标准类库**

1.2 JDK的下载、安装

下载：官网，github

安装：傻瓜式安装：JDK、JRE

注意问题：安装软件的路径中不能包含中文、空格。

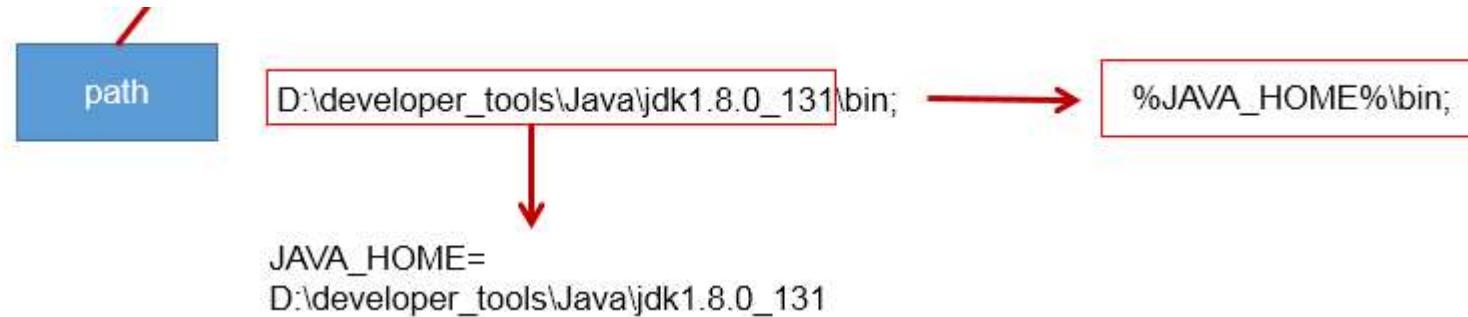
1.3 path环境变量的配置

1.3.1 为什么配置path环境变量？

path环境变量：windows操作系统执行命令时所要搜寻的路径

为什么要配置path：希望java的开发工具（**javac.exe, java.exe**）在任何的文件路径下都可以执行成功。

1.3.2 如何配置？



1. 开发体验—HelloWorld



1.1 编写

创建一个java源文件: HelloWorld.java

```
class HelloChina{
    public static void main(String[] args){
        System.out.println("Hello,World!");
    }
}
```

1.2 编译:

javac HelloWorld.java

1.3 运行:

java HelloChina

2. 常见问题的解决

```
D:\>javac Test1.java
javac: 找不到文件: Test1.java
用法: javac <options> <source files>
-hel p 用于列出可能的选项
```

- 源文件名不存在或者写错
- 当前路径错误
- 后缀名隐藏问题

```
D:\>java Test1
错误: 找不到或无法加载主类 Test1
```

- 类文件名写错，尤其文件名与类名不一致时，要小心
- 类文件不在当前路径下，或者不在classpath指定路径下

```
D:\>javac Test.java
Test.java:1: 错误: 类Test1是公共的, 应在名为 Test1.java 的文件中声明
public class Test1<
               ^
1 个错误
```

- 声明为public的类应与文件名一致，否则编译失败

```
D:\>javac Test.java
Test.java:3: 错误: 需要';
    System.out.println("hello")^
1 个错误
```

- 编译失败，注意错误出现的行数，再到源代码中指定位置改错

3. 总结第一个程序

1. java程序编写-编译-运行的过程

编写：我们将编写的java代码保存在以".java"结尾的源文件中

编译：使用javac.exe命令编译我们的java源文件。格式：javac 源文件名.java

运行：使用java.exe命令解释运行我们的字节码文件。 格式：java 类名

2.

在一个java源文件中可以声明多个class。但是，只能最多有一个类声明为public的。

而且要求声明为public的类的类名必须与源文件名相同。

3. 程序的入口是main()方法。格式是固定的。

4. 输出语句：

System.out.println():先输出数据，然后换行

System.out.print():只输出数据

5. 每一行执行语句都以";"结束。

6. 编译的过程：编译以后，会生成一个或多个字节码文件。字节码文件的文件名与java源文件中的类名相同。

1. 注释: Comment

分类:

单行注释: //

多行注释: /* */

文档注释: /** */

作用:

① 对所写的程序进行解释说明，增强可读性。方便自己，方便别人

② 调试所写的代码

特点:

① 单行注释和多行注释，注释了的内容不参与编译。

换句话说，编译以后生成的.class结尾的字节码文件中不包含注释掉的信息

② 注释内容可以被JDK提供的工具 javadoc 所解析，生成一套以网页文件形式体现的该程序的说明文档。

③ 多行注释不可以嵌套使用

2. Java API 文档:

API: application programming interface。习惯上：将语言提供的类库，都称为api.

API文档：针对于提供的类库如何使用，给的一个说明书。类似于《新华字典》

3. 良好的编程风格

- 正确的注释和注释风格

- 使用文档注释来注释整个类或整个方法。
- 如果注释方法中的某一个步骤，使用单行或多行注释。

- 正确的缩进和空白

- 使用一次tab操作，实现缩进
- 运算符两边习惯性各加一个空格。比如：2 + 4 * 5。

- 块的风格

- Java API 源代码选择了行尾风格

```
public class Test {
    public static void main(String[] args){
        System.out.println("Block Style!");
    }
}
```

行尾风格

```
public class Test
{
    public static void main(String[] args)
    {
        System.out.println("Block Style!");
    }
}
```

次行风格

让天下没有难学的技术

1. 开发工具说明：

● 文本编辑工具：

- 记事本
- UltraEdit
- EditPlus
- TextPad
- NotePad

● Java集成开发环境 (IDE)：

- JBuilder
- NetBeans
- Eclipse
- MyEclipse
- IntelliJ IDEA

2. EditPlus的使用：



1. java关键字的使用

定义：被Java语言赋予了特殊含义，用做专门用途的字符串（单词）

特点：关键字中所字母都为小写

具体哪些关键字：

用于定义数据类型的关键字				
class	interface	enum	byte	short
int	long	float	double	char
boolean	void			
用于定义流程控制的关键字				
if	else	switch	case	default
while	do	for	break	continue
return				
用于定义访问权限修饰符的关键字				
private	protected	public		

用于定义类，函数，变量修饰符的关键字				
abstract	final	static	synchronized	
用于定义类与类之间关系的关键字				
extends	implements			
用于定义建立实例及引用实例，判断实例的关键字				
new	this	super	instanceof	
用于异常处理的关键字				
try	catch	finally	throw	throws
用于包的关键字				
package	import			
其他修饰符关键字				
native	strictfp	transient	volatile	assert
* 用于定义数据类型值的字面值				
true	false	null		

2. 保留字： 现Java版本尚未使用，但以后版本可能会作为关键字使用。

具体哪些保留字： goto 、 const

注意：自己命名标识符时要避免使用这些保留字

3. 标识符的使用

定义：凡是自己可以起名字的地方都叫标识符。

涉及到的结构：

包名、类名、接口名、变量名、方法名、常量名

规则：(必须要遵守。否则，编译不通过)

- 由26个英文字母大小写，0-9，_或\$组成
- 数字不可以开头。
- 不可以使用关键字和保留字，但能包含关键字和保留字。
- Java中严格区分大小写，长度无限制。
- 标识符不能包含空格。

规范：（可以不遵守，不影响编译和运行。但是要求大家遵守）

- **包名**：多单词组成时所有字母都小写：xxxxyyyzzz
- **类名、接口名**：多单词组成时，所有单词的首字母大写：XxxYyyZzz
- **变量名、方法名**：多单词组成时，第一个单词首字母小写，第二个单词开始每个单词首字母大写：xxxYyyZzz
- **常量名**：所有字母都大写。多单词时每个单词用下划线连接：XXX_YYY_ZZZ

注意点：

- 在起名字时，为了提高阅读性，要尽量意义，“见名知意”。

代码整洁之道

整理人：尚硅谷 - 宋红康

第2章 有意义的命名

2.1 介绍

软件中随处可见命名。我们给变量、函数、参数、类和包命名。我们给源代码及源代码所在目录命名。

这么多命名要做，不妨做好它。下文列出了取个好名字的几条简单规则。

2.2 名副其实，见名知意

变量名太随意，haha、list1、ok、theList 这些都没啥意义

2.3 避免误导

包含List、import、java等类名、关键字或特殊字；

字母o与数字0，字母1与数字1等

提防使用不同之处较小的名称。比如：XYZControllerForEfficientHandlingOfStrings与XYZControllerForEfficientStorageOfStrings

2.4 做有意义的区分

反面教材，变量名：a1、a2、a3

避免冗余，不要出现Variable、表字段中避免出现table、字符串避免出现nameString，直接name就行，知道是字符串类型

再比如：定义了两个类：Customer类和CustomerObject类，如何区分？

定义了三个方法：getActiveAccount()、getActiveAccounts()、getActiveAccountInfo()，如何区分？

2.5 使用读得出来的名称

不要使用自己拼凑出来的单词，比如：xsxm(学生姓名)；genymdhms(生成日期，年、月、日、时、分、秒)

所谓的驼峰命名法，尽量使用完整的单词

2.6 使用可搜索的名称

一些常量，最好不直接使用数字，而指定一个变量名，这个变量名可以便于搜索到。

比如：找MAX_CLASSES_PER_STUDENT很容易，但想找数字7就麻烦了。

2.7 避免使用编码

2.7.1 匈牙利语标记法

即变量名表明该变量数据类型的小写字母开始。例如，szCmdLine的前缀sz表示“以零结束的字符串”。

2.7.2 成员前缀

避免使用前缀，但是Android中一个比较好的喜欢用m表示私有等，个人感觉比较好

2.7.3 接口和实现

作者不喜欢把接口使用I来开头，实现也希望只是在后面添加Imp

2.8 避免思维映射

比如传统上惯用单字母名称做循环计数器。所以就不要给一些非计数器的变量命名为：i、j、k等

2.9 类名

类名与对象名应该是名词与名词短语。如Customer、WikiPage、Account和AddressParser。避免使用Data或Info这样的类名。

不能使动词。比如：Manage、Process

2.10 方法名

方法名应当是动词或者动词短语。如postPayment、deletePage或save

2.11 别扮可爱

有的变量名叫haha、banana

别用eatMyShorts()表示abort()

2.12 每个概念对应一个词

项目中同时出现controllers与managers，为什么不统一使用其中一种？

对于那些会用到你代码的程序员，一以贯之的命名法简直就是天降福音。

2.13 别用双关语

有时可能使用add并不合适，比例insert、append。add表示完整的新添加的含义。

2.14 使用解决方案领域名称

看代码的都是程序员，所以尽量用那些计算机科学术语、算法名、模式名、数学术语，

依据问题所涉领域来命名不算是聪明的做法。

2.15 使用源自所涉问题领域的名称

如果不能用程序员熟悉的术语来给手头的工作命名，就采用从所涉问题领域而来的名称吧。

至少，负责维护代码的程序员就能去请教领域专家了。

2.16 添加有意义的语境

可以把相关的变量放到一个类中，使用这个类来表明语境。

2.17 不要添加没用的语境

名字中带有项目的缩写，这样完全没有必要。比如有一个名为“加油站豪华版”（Gas Station Deluxe）的项目，

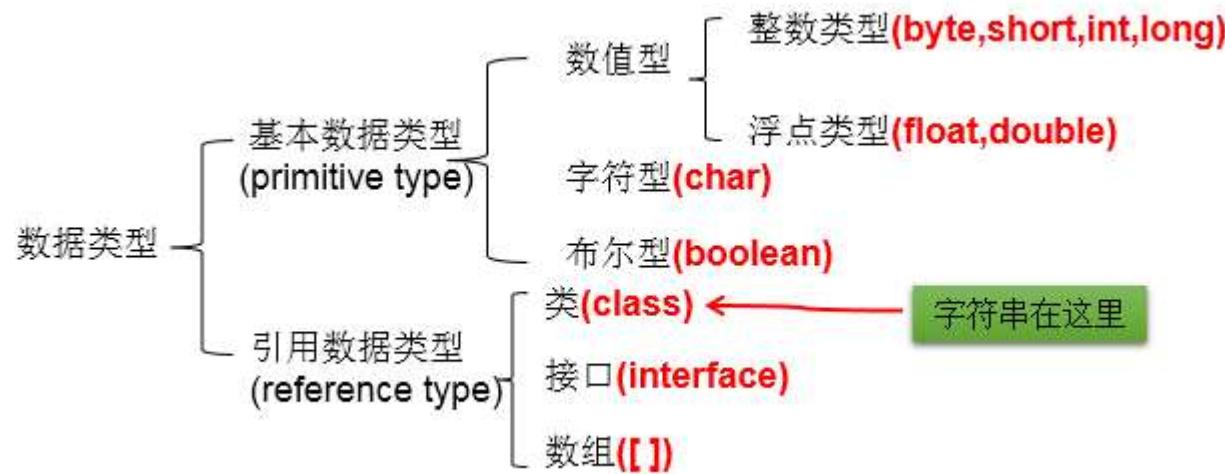
在其中给每个类添加GSD前缀就不是什么好策略。

2.18 最后的话

取好名字最难的地方在于需要良好的描述技巧和共有文化背景。

1. 变量的分类

1.1 按数据类型分类



详细说明：

//1. 整型： byte(1字节=8bit) \ short(2字节) \ int(4字节) \ long(8字节)

//① byte范围： -128 ~ 127

// ② 声明long型变量，必须以"l"或"L"结尾

// ③ 通常，定义整型变量时，使用int型。

//④ 整型的常量，默认类型是： int型

//2. 浮点型： float(4字节) \ double(8字节)

//① 浮点型，表示带小数点的数值

//② float表示数值的范围比long还大

//③ 定义float类型变量时，变量要以"f"或"F"结尾

//④ 通常，定义浮点型变量时，使用double型。

//⑤ 浮点型的常量，默认类型为： double

//3. 字符型： char (1字符=2字节)

//① 定义char型变量，通常使用一对'', 内部只能写一个字符

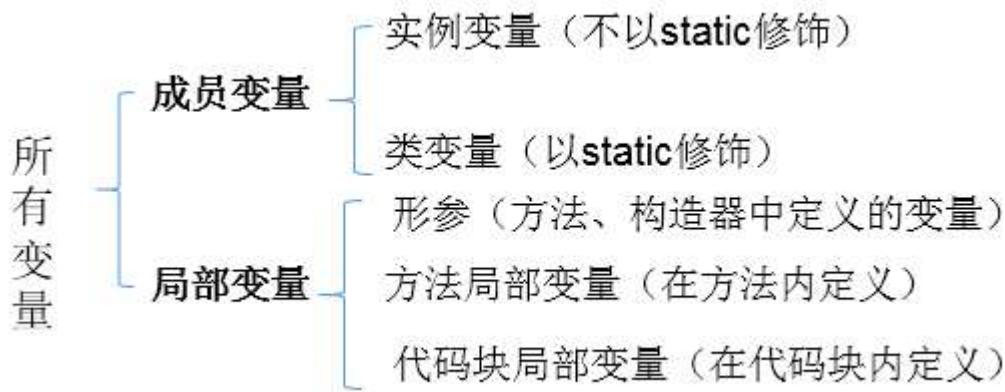
//② 表示方式： 1. 声明一个字符 2. 转义字符 3. 直接使用 Unicode 值来表示字符型常量

//4. 布尔型： boolean

//① 只能取两个值之一： true 、 false

//② 常常在条件判断、循环结构中使用

1.2 按声明的位置分类(了解)



2. 定义变量的格式:

数据类型 变量名 = 变量值;

或

数据类型 变量名;

变量名 = 变量值;

3. 变量使用的注意点:

- ① 变量必须先声明，后使用
- ② 变量都定义在其作用域内。在作用域内，它是有效的。换句话说，出了作用域，就失效了
- ③ 同一个作用域内，不可以声明两个同名的变量

4. 基本数据类型变量间运算规则

4.1 涉及到的基本数据类型：除了boolean之外的其他7种

4.2 自动类型转换(只涉及7种基本数据类型)

结论：当容量小的数据类型的变量与容量大的数据类型的变量做运算时，结果自动提升为容量大的数据类型。

`byte`、`char`、`short` --> `int` --> `long` --> `float` --> `double`

特别的：当`byte`、`char`、`short`三种类型的变量做运算时，结果为`int`型

说明：此时的容量大小指的是，表示数的范围的大和小。比如：`float`容量要大于`long`的容量

4.3 强制类型转换(只涉及7种基本数据类型)：自动类型提升运算的逆运算。

1. 需要使用强转符：()
2. 注意点：强制类型转换，可能导致精度损失。

4.4 String与8种基本数据类型间的运算

1. `String`属于引用数据类型，翻译为：字符串
2. 声明`String`类型变量时，使用一对""
3. `String`可以和8种基本数据类型变量做运算，且运算只能是连接运算：+
4. 运算的结果仍然是`String`类型

避免:

```
String s = 123;//编译错误
```

```
String s1 = "123";
```

```
int i = (int)s1;//编译错误
```

1. 编程中涉及的进制及表示方式:

- **二进制(binary):** 0,1 , 满2进1.以**0b或0B**开头。
- **十进制(decimal):** 0-9 , 满10进1。
- **八进制(octal):** 0-7 , 满8进1. 以数字**0开头**表示。
- **十六进制(hex):** 0-9及A-F, 满16进1. 以**0x或0X开头**表示。此处的A-F不区分大小写。
如: $0x21AF + 1 = 0X21B0$

2. 二进制的使用说明:

2.1 计算机底层的存储方式: 所有数字在计算机底层都以**二进制**形式存在。

2.2 二进制数据的存储方式: 所有的数值, 不管正负, 底层都以补码的方式存储。

2.3 原码、反码、补码的说明:

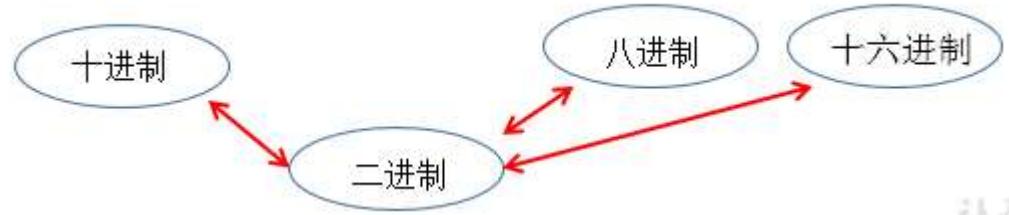
正数: 三码合一

负数:

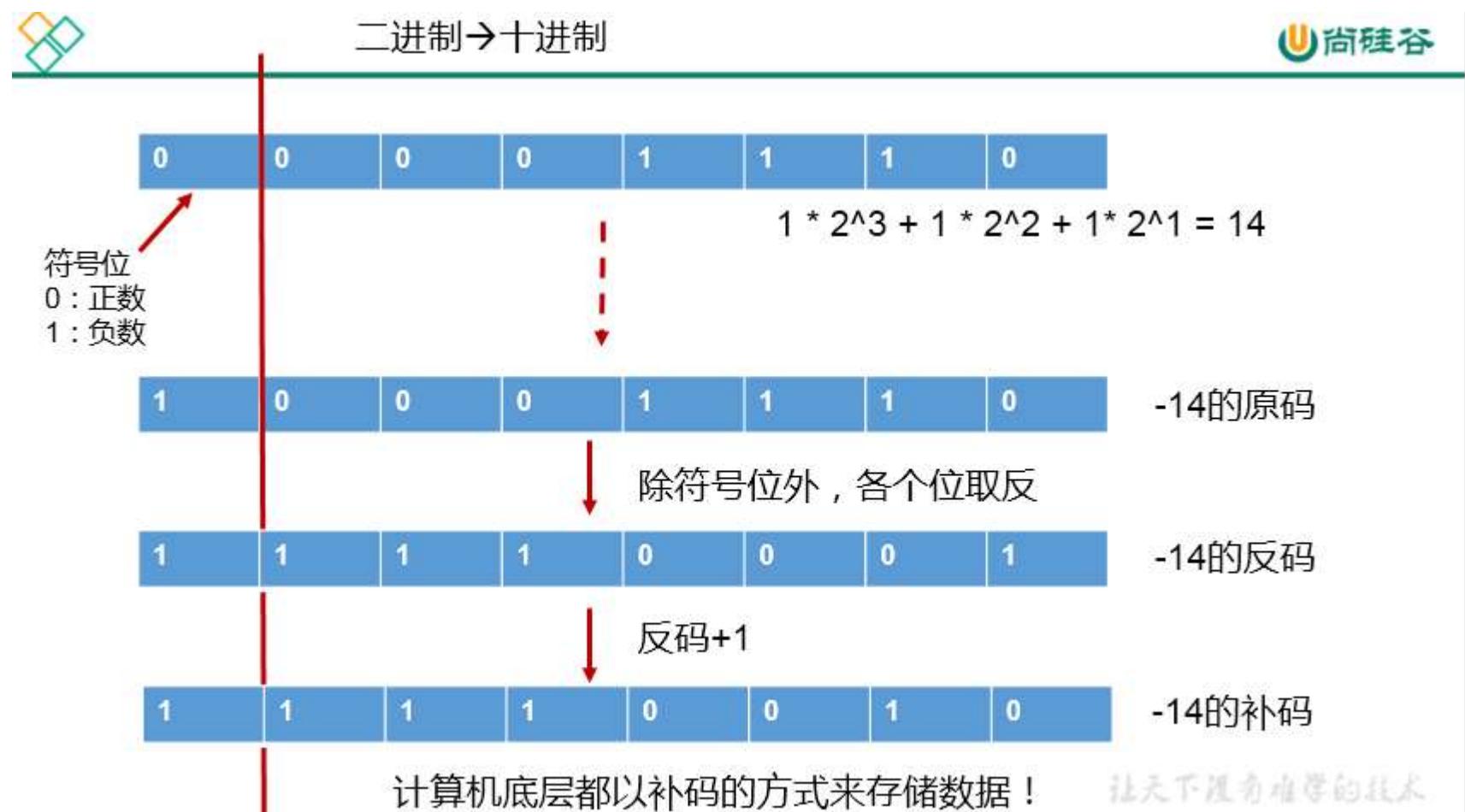
- **原码:** 直接将一个数值换成二进制数。最高位是符号位
- **负数的反码:** 是对原码按位取反, 只是最高位(符号位)确定为1。
- **负数的补码:** 其反码加1。

3. 进制间的转换:

3.1 图示:



3.2 图示二进制转换为十进制:





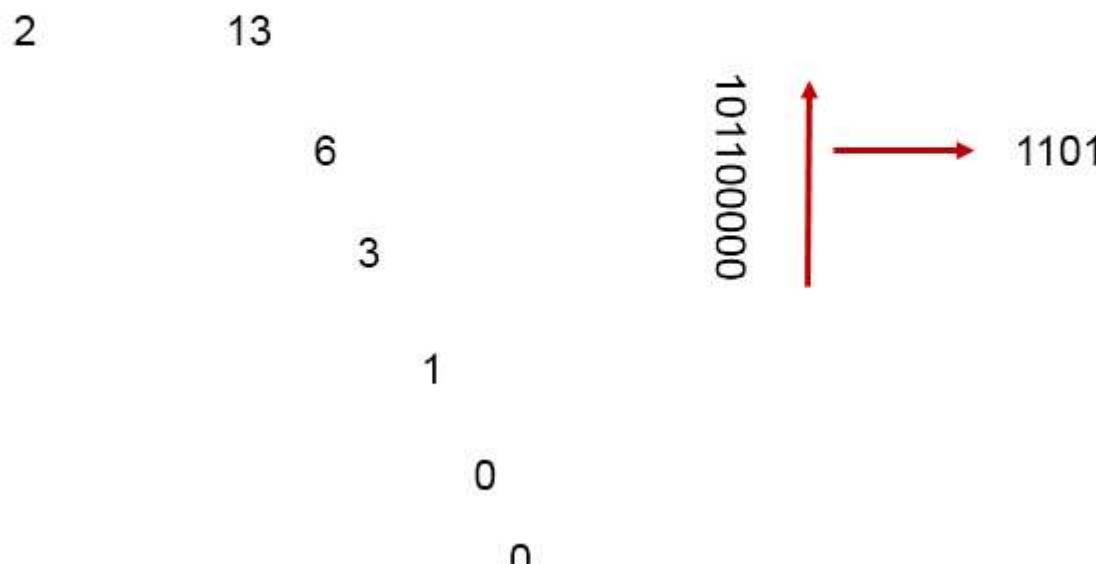
0	1	1	1	1	1	1	1	+127
1	1	1	1	1	1	1	1	-127的原码
1	0	0	0	0	0	0	0	-127的反码
1	0	0	0	0	0	0	1	-127的补码
1	0	0	0	0	0	0	0	-128的补码

让天下没有难学的技术

3.3 图示十进制转换为二进制：

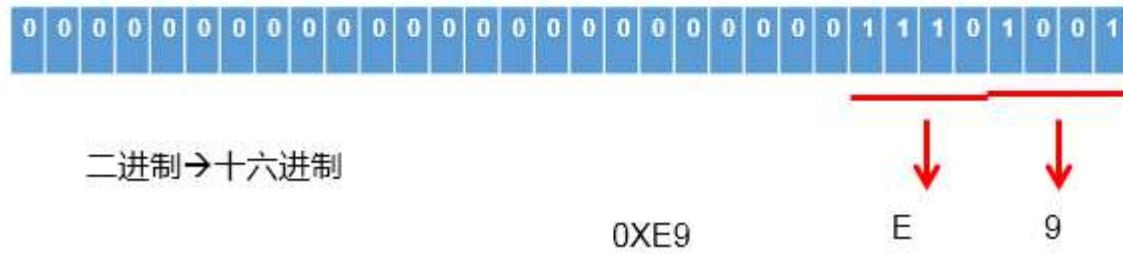
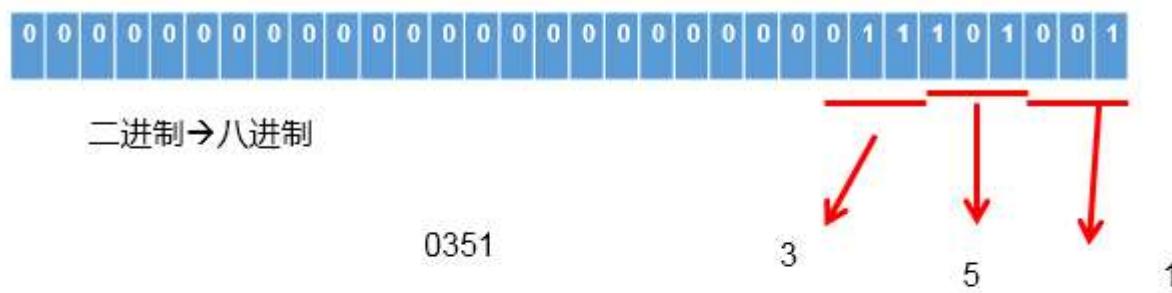


十进制 → 二进制：除2取余的逆



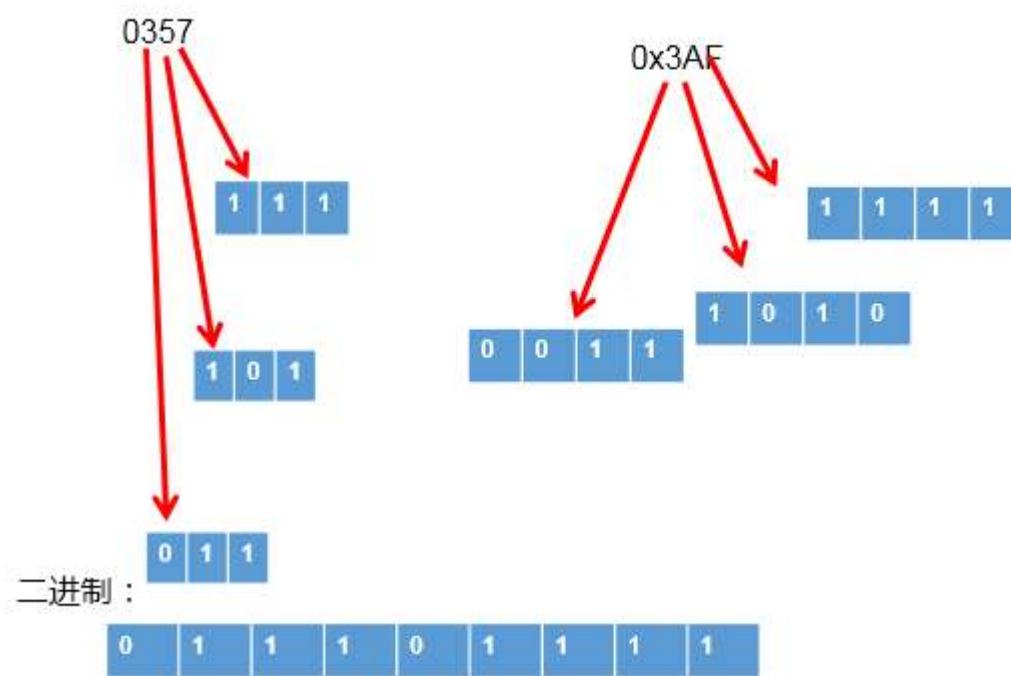
3.4 二进制与八进制、十六进制间的转换：

111→7



八进制：

十六进制



运算符

1-算术运算符

1. 算术运算符: + - + - * / % (前)++ (后)++ (前)-- (后)-- +

【典型代码】

```
//除号: /  
int num1 = 12;  
int num2 = 5;  
int result1 = num1 / num2;  
System.out.println(result1); //2  
  
// %:取余运算  
//结果的符号与被模数的符号相同  
//开发中, 经常使用%来判断能否被除尽的情况。  
int m1 = 12;  
int n1 = 5;  
System.out.println("m1 % n1 = " + m1 % n1);  
  
int m2 = -12;  
int n2 = 5;  
System.out.println("m2 % n2 = " + m2 % n2);  
  
int m3 = 12;  
int n3 = -5;  
System.out.println("m3 % n3 = " + m3 % n3);  
  
int m4 = -12;  
int n4 = -5;  
System.out.println("m4 % n4 = " + m4 % n4);  
//(前)++ :先自增1, 后运算  
//(后)++ :先运算, 后自增1  
int a1 = 10;  
int b1 = ++a1;  
System.out.println("a1 = " + a1 + ", b1 = " + b1);  
  
int a2 = 10;  
int b2 = a2++;  
System.out.println("a2 = " + a2 + ", b2 = " + b2);  
  
int a3 = 10;
```

```
++a3;//a3++;  
int b3 = a3;  
//(前)-- :先自减1, 后运算  
//(后)-- :先运算, 后自减1  
  
int a4 = 10;  
int b4 = a4--;//int b4 = --a4;  
System.out.println("a4 = " + a4 + ",b4 = " + b4);
```

【特别说明的】

- 1.//(前)++ :先自增1, 后运算
//(后)++ :先运算, 后自增1
- 2.//(前)-- :先自减1, 后运算
//(后)-- :先运算, 后自减1
- 3.连接符: +: 只能使用在String与其他数据类型变量之间使用。

2. 赋值运算符: = += -= *= /= %=

【典型代码】

```
int i2,j2;
//连续赋值
i2 = j2 = 10;
/*************
int i3 = 10,j3 = 20;
int num1 = 10;
num1 += 2;//num1 = num1 + 2;
System.out.println(num1);//12

int num2 = 12;
num2 %= 5;//num2 = num2 % 5;
System.out.println(num2);

short s1 = 10;
//s1 = s1 + 2;//编译失败
s1 += 2;//结论: 不会改变变量本身的数据类型
System.out.println(s1);
```

【特别说明的】

1. 运算的结果不会改变变量本身的数据类型

2.

//开发中, 如果希望变量实现+2的操作, 有几种方法? (前提: int num = 10;)

//方式一: num = num + 2;

//方式二: num += 2; (推荐)

//开发中, 如果希望变量实现+1的操作, 有几种方法? (前提: int num = 10;)

//方式一: num = num + 1;

//方式二: num += 1;

//方式三: num++; (推荐)

3. 比较运算符（关系运算符）： == != > < >= <= instanceof

【典型代码】

```
int i = 10;  
int j = 20;  
  
System.out.println(i == j); //false  
System.out.println(i = j); //20  
  
boolean b1 = true;  
boolean b2 = false;  
System.out.println(b2 == b1); //false  
System.out.println(b2 = b1); //true
```

【特别说明的】

1. 比较运算符的结果是boolean类型

2. > < >= <= : 只能使用在数值类型的数据之间。

3. == 和 !=: 不仅可以使用在数值类型数据之间，还可以使用在其他引用类型变量之间。

```
Account acct1 = new Account(1000);  
Account acct2 = new Account(1000);  
boolean b1 = (acct1 == acct2); //比较两个Account是否是同一个账户。  
boolean b2 = (acct1 != acct2); //
```

4. 逻辑运算符: & && | || ! ^

【典型代码】

```
//区分& 与 &&
//相同点1: & 与 && 的运算结果相同
//相同点2: 当符号左边是true时, 二者都会执行符号右边的运算
//不同点: 当符号左边是false时, &继续执行符号右边的运算。&&不再执行符号右边的运算。
//开发中, 推荐使用&&

boolean b1 = true;
b1 = false;
int num1 = 10;
if(b1 & (num1++ > 0)){
    System.out.println("我现在在北京");
}else{
    System.out.println("我现在在南京");
}

System.out.println("num1 = " + num1);

boolean b2 = true;
b2 = false;
int num2 = 10;
if(b2 && (num2++ > 0)){
    System.out.println("我现在在北京");
}else{
    System.out.println("我现在在南京");
}

System.out.println("num2 = " + num2);

// 区分: | 与 ||
//相同点1: | 与 || 的运算结果相同
//相同点2: 当符号左边是false时, 二者都会执行符号右边的运算
//不同点3: 当符号左边是true时, |继续执行符号右边的运算, 而||不再执行符号右边的运
算
```

```
//开发中，推荐使用||

boolean b3 = false;

b3 = true;

int num3 = 10;

if(b3 || (num3++ > 0)) {
    System.out.println("我现在在北京");
} else {
    System.out.println("我现在在南京");
}

System.out.println("num3 = " + num3);
```

```
boolean b4 = false;

b4 = true;

int num4 = 10;

if(b4 || (num4++ > 0)) {
    System.out.println("我现在在北京");
} else {
    System.out.println("我现在在南京");
}

System.out.println("num4 = " + num4);
```

【特别说明的】

1. 逻辑运算符操作的都是boolean类型的变量。而且结果也是boolean类型

5.位运算符: << >> >>> & | ^ ~

【典型代码】

```
int i = 21;  
i = -21;  
System.out.println("i << 2 :" + (i << 2));  
System.out.println("i << 3 :" + (i << 3));  
System.out.println("i << 27 :" + (i << 27));  
  
int m = 12;  
int n = 5;  
System.out.println("m & n :" + (m & n));  
System.out.println("m | n :" + (m | n));  
System.out.println("m ^ n :" + (m ^ n));
```

【面试题】 你能否写出最高效的 $2 * 8$ 的实现方式?

答案: $2 \ll 3$ 或 $8 \ll 1$

【特别说明的】

1. 位运算符操作的都是整型的数据
2. << : 在一定范围内, 每向左移1位, 相当于 $* 2$
>> : 在一定范围内, 每向右移1位, 相当于 $/ 2$

典型题目:

1. 交换两个变量的值。
2. 实现60的二进制到十六进制的转换

6. 三元运算符：(条件表达式)? 表达式1 : 表达式2

【典型代码】

1. 获取两个整数的较大值

2. 获取三个数的最大值

【特别说明的】

1. 说明

① 条件表达式的结果为boolean类型

② 根据条件表达式真或假，决定执行表达式1，还是表达式2。

如果表达式为true，则执行表达式1。

如果表达式为false，则执行表达式2。

③ 表达式1 和表达式2要求是一致的。

④ 三元运算符可以嵌套使用

2.

凡是可以说使用三元运算符的地方，都可以改写为if-else

反之，不成立。

3. 如果程序既可以使用三元运算符，又可以使用if-else结构，那么优先选择三元运算符。原因：简洁、执行效率高。

流程控制

顺序结构：程序从上到下执行。

分支结构：

`if-else if - else`

`switch-case`

循环结构：

`for`

`while`

`do-while`

分支结构

1. if-else条件判断结构**1.1.****结构一:**

```
if(条件表达式){  
    执行表达式  
}
```

结构二: 二选一

```
if(条件表达式){  
    执行表达式1  
}else{  
    执行表达式2  
}
```

结构三: n选一

```
if(条件表达式){  
    执行表达式1  
}else if(条件表达式){  
    执行表达式2  
}else if(条件表达式){  
    执行表达式3  
}  
...  
else{  
    执行表达式n  
}
```

1.2.说明:

1. **else** 结构是可选的。

2. 针对于条件表达式:

- > 如果多个条件表达式之间是“互斥”关系(或没有交集的关系),哪个判断和执行语句声明在上面还是下

面，无所谓。

› 如果多个条件表达式之间有交集的关系，需要根据实际情况，考虑清楚应该将哪个结构声明在上面。

› 如果多个条件表达式之间有包含的关系，通常情况下，需要将范围小的声明在范围大的上面。否则，范围小的就没机会执行了。

3. **if-else**结构是可以相互嵌套的。

4. 如果**if-else**结构中的执行语句只有一行时，对应的一对{}可以省略的。但是，不建议大家省略。

2. switch-case选择结构

```
switch(表达式){
    case 常量1:
        执行语句1;
        //break;
    case 常量2:
        执行语句2;
        //break;
    ...
    default:
        执行语句n;
        //break;
}
```

2. 说明：

① 根据**switch**表达式中的值，依次匹配各个**case**中的常量。一旦匹配成功，则进入相应**case**结构中，调用其执行语句。

当调用完执行语句以后，则仍然继续向下执行其他**case**结构中的执行语句，直到遇到**break**关键字或此**switch-case**结构

末尾结束为止。

② **break**, 可以使用在**switch-case**结构中，表示一旦执行到此关键字，就跳出**switch-case**结构

③ **switch**结构中的表达式，只能是如下的6种数据类型之一：

byte、**short**、**char**、**int**、枚举类型(JDK5.0新增)、**String**类型(JDK7.0新增)

④ **case** 之后只能声明常量。不能声明范围。

⑤ **break**关键字是可选的。

⑥ **default**:相当于**if-else**结构中的**else**.

default结构是可选的，而且位置是灵活的。

3. 如果**switch-case**结构中的多个**case**的执行语句相同，则可以考虑进行合并。

4. **break**在**switch-case**中是可选的

循环结构

1. 循环结构的四要素

- ① 初始化条件
- ② 循环条件 ---> 是boolean类型
- ③ 循环体
- ④ 迭代条件

说明：通常情况下，循环结束都是因为②中循环条件返回false了。

2. 三种循环结构：

2.1 for循环结构

```
for(①;②;④){
```

 ③

}

执行过程： ① - ② - ③ - ④ - ② - ③ - ④ - ... - ②

2.2 while循环结构

①

```
while(②){
```

 ③;

 ④;

}

执行过程： ① - ② - ③ - ④ - ② - ③ - ④ - ... - ②

说明：

写while循环千万小心不要丢了迭代条件。一旦丢了，就可能导致死循环！

for和while循环总结：

1. 开发中，基本上我们都会从for、while中进行选择，实现循环结构。
2. for循环和while循环是可以相互转换的！

区别： for循环和while循环的初始化条件部分的作用范围不同。

3. 我们写程序，要避免出现死循环。

2.3 do-while循环结构

①

```
do{
```

```

③;
④;
}while(②);

```

执行过程: ① - ③ - ④ - ② - ③ - ④ - ... - ②

说明:

1. do-while循环至少会执行一次循环体!
2. 开发中, 使用for和while更多一些。较少使用do-while

3.“无限循环”结构: while(true) 或 for(;;)

总结: 如何结束一个循环结构?

方式一: 当循环条件是false时

方式二: 在循环体中, 执行break

4. 嵌套循环

1. 嵌套循环: 将一个循环结构A声明在另一个循环结构B的循环体中, 就构成了嵌套循环

内层循环: 循环结构A

外层循环: 循环结构B

2. 说明:

- ① 内层循环结构遍历一遍, 只相当于外层循环循环体执行了一次
- ② 假设外层循环需要执行m次, 内层循环需要执行n次。此时内层循环的循环体一共执行了m * n次
- ③ 外层循环控制行数, 内层循环控制列数

【典型练习】

```

//练习一:

/*
*****
*****
*****
*/
for(int j = 1;j <= 4;){j++
    for(int i = 1;i <= 6;i++){
        System.out.print('*');
    }
}

```

```
System.out.println();  
}  
  
//练习二:  
  
/*          i(行号)          j(*的个数)  
 *           1                  1  
 **          2                  2  
 ***         3                  3  
 ****        4                  4  
 *****       5                  5  
 */  
  
  
for(int i = 1;i <= 5;i++){//控制行数  
    for(int j = 1;j <= i;j++){//控制列数  
        System.out.print("*");  
    }  
    System.out.println();  
}  
  
//练习三: 九九乘法表  
//练习四: 100以内的质数
```

补充:衡量一个功能代码的优劣:

1. 正确性
2. 可读性
3. 健壮性
4. 高效率与低存储: **时间复杂度**、空间复杂度 (衡量算法的好坏)

如何理解流程控制的练习:

流程控制结构的使用 + 算法逻辑

关键字: break和continue

break和continue关键字的使用

	使用范围	循环中使用的作用(不同点)	相同点
break:	switch-case		
	循环结构中	结束当前循环	关键字后面不能声明执行语句

continue:	循环结构中	结束当次循环	关键字后面不能声明执行语句
------------------	-------	--------	---------------

补充: 带标签的break和continue的使用

return在方法中讲。

补充：Scanner类的使用

```
/*
```

如何从键盘获取不同类型的变量：需要使用Scanner类

具体实现步骤：

1. 导包：import java.util.Scanner;
2. Scanner的实例化：Scanner scan = new Scanner(System.in);
3. 调用Scanner类的相关方法（next() / nextXxx()），来获取指定类型的变量

注意：

需要根据相应的方法，来输入指定类型的值。如果输入的数据类型与要求的类型不匹配时，会报异常：InputMismatchException
导致程序终止。

```
*/
```

```
//1. 导包：import java.util.Scanner;
import java.util.Scanner;
```

```
class ScannerTest{
```

```
    public static void main(String[] args) {
        //2. Scanner的实例化
        Scanner scan = new Scanner(System.in);

        //3. 调用Scanner类的相关方法
        System.out.println("请输入你的姓名：");
        String name = scan.next();
        System.out.println(name);
```

```
        System.out.println("请输入你的年龄：");
        int age = scan.nextInt();
        System.out.println(age);
```

```
        System.out.println("请输入你的体重：");
        double weight = scan.nextDouble();
        System.out.println(weight);
```

```
        System.out.println("你是否相中我了呢？(true/false)");
        boolean isLove = scan.nextBoolean();
        System.out.println(isLove);
```

```
//对于char型的获取，Scanner没有提供相关的方法。只能获取一个字符串
```

```
        System.out.println("请输入你的性别：(男/女)");
        String gender = scan.next(); //男
        char genderChar = gender.charAt(0); //获取索引为0位置上的字符
        System.out.println(genderChar);
```

```
}
```

数组的概述

- * 1. 数组的理解：数组(Array)，是多个相同类型数据一定顺序排列的集合，并使用一个名字命名，
 - * 并通过编号的方式对这些数据进行统一管理。
 - *
- * 2. 数组相关的概念：
 - * >数组名
 - * >元素
 - * >角标、下标、索引
 - * >数组的长度：元素的个数
 - *
- * 3. 数组的特点：
 - * 1数组是序排列的
 - * 2数组属于引用数据类型的变量。数组的元素，既可以是基本数据类型，也可以是引用数据类型
 - * 3创建数组对象会在内存中开辟一整块连续的空间
 - * 4数组的长度一旦确定，就不能修改。
 - *
- * 4. 数组的分类：
 - * ① 照维数：一维数组、二维数组、。。。
 - * ② 照数组元素的类型：基本数据类型元素的数组、引用数据类型元素的数组

数据结构：

1. 数据与数据之间的逻辑关系：集合、一对一、一对多、多对多

2. 数据的存储结构：

线性表：顺序表（比如：数组）、链表、栈、队列

树形结构：二叉树

图形结构：

算法：

排序算法：

搜索算法：

1. 一维数组的声明与初始化

正确的方式：

```
int num; // 声明
num = 10; // 初始化
int id = 1001; // 声明 + 初始化
```

```
int[] ids; // 声明
// 1.1 静态初始化：数组的初始化和数组元素的赋值操作同时进行
ids = new int[] {1001, 1002, 1003, 1004};
// 1.2 动态初始化：数组的初始化和数组元素的赋值操作分开进行
String[] names = new String[5];
```

```
int[] arr4 = {1, 2, 3, 4, 5}; // 类型推断
```

错误的方式：

```
// int[] arr1 = new int[];
// int[5] arr2 = new int[5];
// int[] arr3 = new int[3] {1, 2, 3};
```

2. 一维数组元素的引用：通过角标的方式调用。

```
// 数组的角标（或索引从0开始的，到数组的长度-1结束。
names[0] = "王铭";
names[1] = "王赫";
names[2] = "张学良";
names[3] = "孙居龙";
names[4] = "王宏志"; // charAt(0)
```

3. 数组的属性：length

```
System.out.println(names.length); // 5
System.out.println(ids.length);
```

说明：

数组一旦初始化，其长度就是确定的。arr.length
数组长度一旦确定，就不可修改。

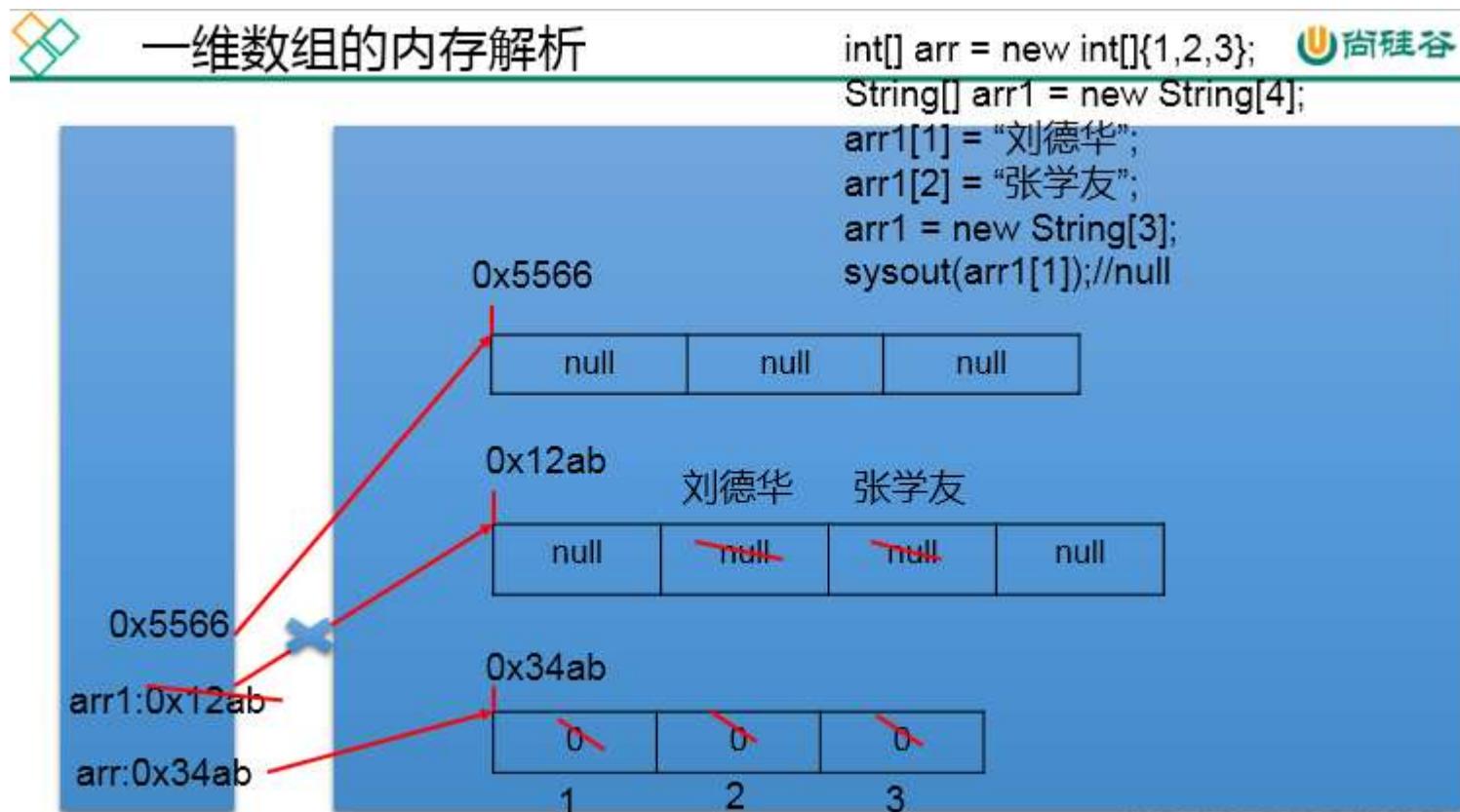
4. 一维数组的遍历

```
for(int i = 0; i < names.length; i++) {
    System.out.println(names[i]);
}
```

5. 一维数组元素的默认初始化值

- * > 数组元素是整型：0
- * > 数组元素是浮点型：0.0
- * > 数组元素是char型：0或'\u0000'，而非'0'
- * > 数组元素是boolean型：false
- *
- * > 数组元素是引用数据类型：null

6. 一维数组的内存解析



二维数组

1. 如何理解二维数组？

数组属于引用数据类型

数组的元素也可以是引用数据类型

一个一维数组A的元素如果还是一个一维数组类型的，则，此数组A称为二维数组。

2. 二维数组的声明与初始化

正确的方式：

```

int[] arr = new int[]{1,2,3}; //一维数组
//静态初始化
int[][] arr1 = new int[][]{{1,2,3},{4,5},{6,7,8}};
//动态初始化1
String[][] arr2 = new String[3][2];
//动态初始化2
String[][] arr3 = new String[3][];
//也是正确的写法：
int[] arr4[] = new int[][]{{1,2,3},{4,5,9,10},{6,7,8}};
int[] arr5[] = {{1,2,3},{4,5},{6,7,8}}; //类型推断

```

错误的方式：

```

//      String[][] arr4 = new String[][][4];
//      String[4][3] arr5 = new String[][][];
//      int[][] arr6 = new int[4][3]{{1,2,3},{4,5},{6,7,8}};

```

3. 如何调用二维数组元素：

```

System.out.println(arr1[0][1]); //2
System.out.println(arr2[1][1]); //null

arr3[1] = new String[4];
System.out.println(arr3[1][0]);
System.out.println(arr3[0]);

```

4. 二维数组的属性：

```

System.out.println(arr4.length); //3
System.out.println(arr4[0].length); //3
System.out.println(arr4[1].length); //4

```

5. 遍历二维数组元素

```

for(int i = 0;i < arr4.length;i++){
    for(int j = 0;j < arr4[i].length;j++){
        System.out.print(arr4[i][j] + " ");
    }
    System.out.println();
}

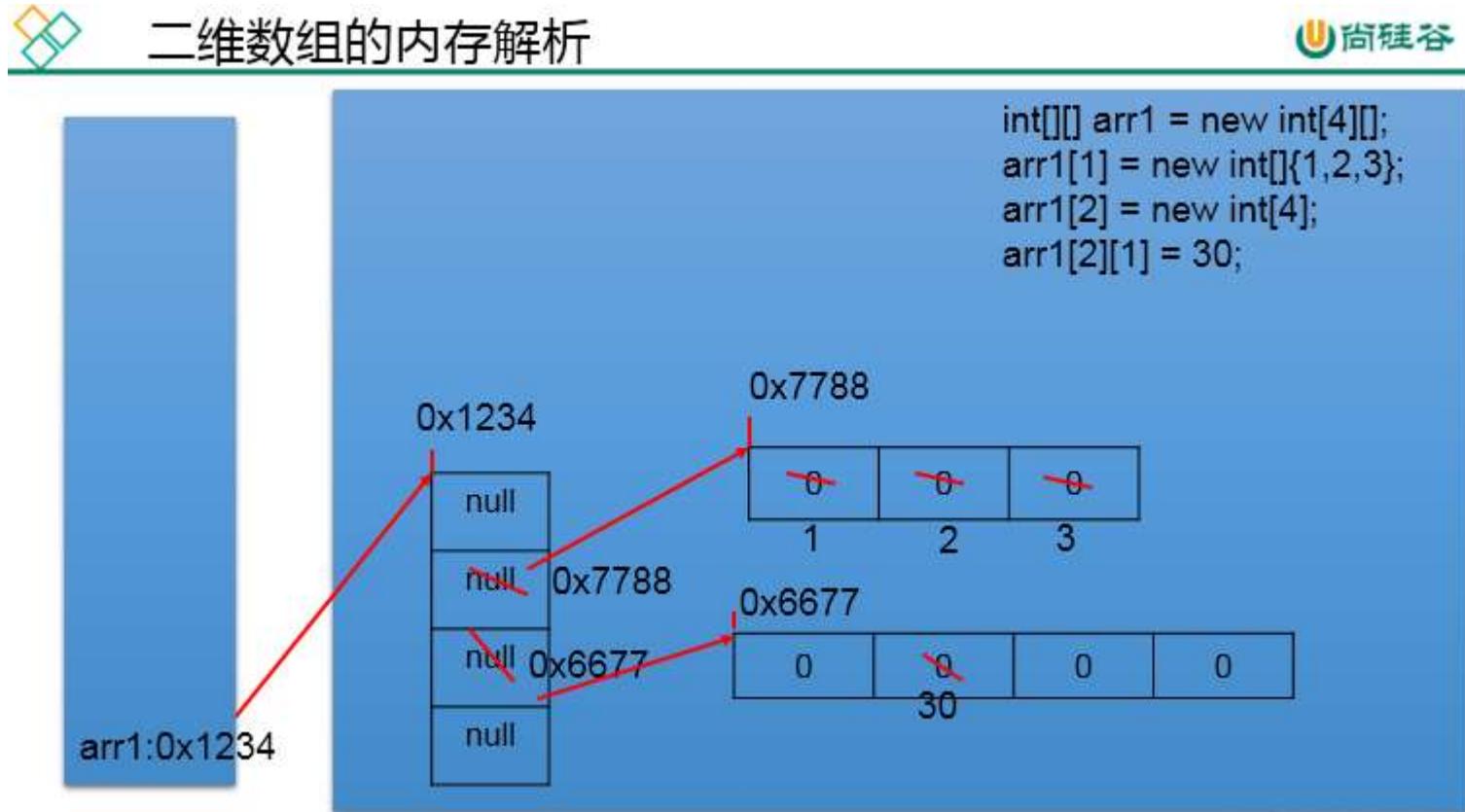
```

6. 二维数组元素的默认初始化值

- * 规定：二维数组分为外层数组的元素，内层数组的元素
- * `int[][] arr = new int[4][3];`
- * 外层元素：`arr[0], arr[1]`等
- * 内层元素：`arr[0][0], arr[1][2]`等
- *
- * ⑤ 数组元素的默认初始化值
- * 针对于初始化方式一：比如：`int[][] arr = new int[4][3];`
- * 外层元素的初始化值为：地址值

- * 内层元素的初始化值为：与一维数组初始化情况相同
- * 针对于初始化方式二：比如：`int[][] arr = new int[4][];`
- * 外层元素的初始化值为：`null`
- * 内层元素的初始化值为：不能调用，否则报错。

7. 二维数组的内存结构



1. 数组的创建与元素赋值：

杨辉三角（二维数组）、回形数（二维数组）、6个数，1-30之间随机生成且不重复。

2. 针对于数值型的数组：

最大值、最小值、总和、平均数等

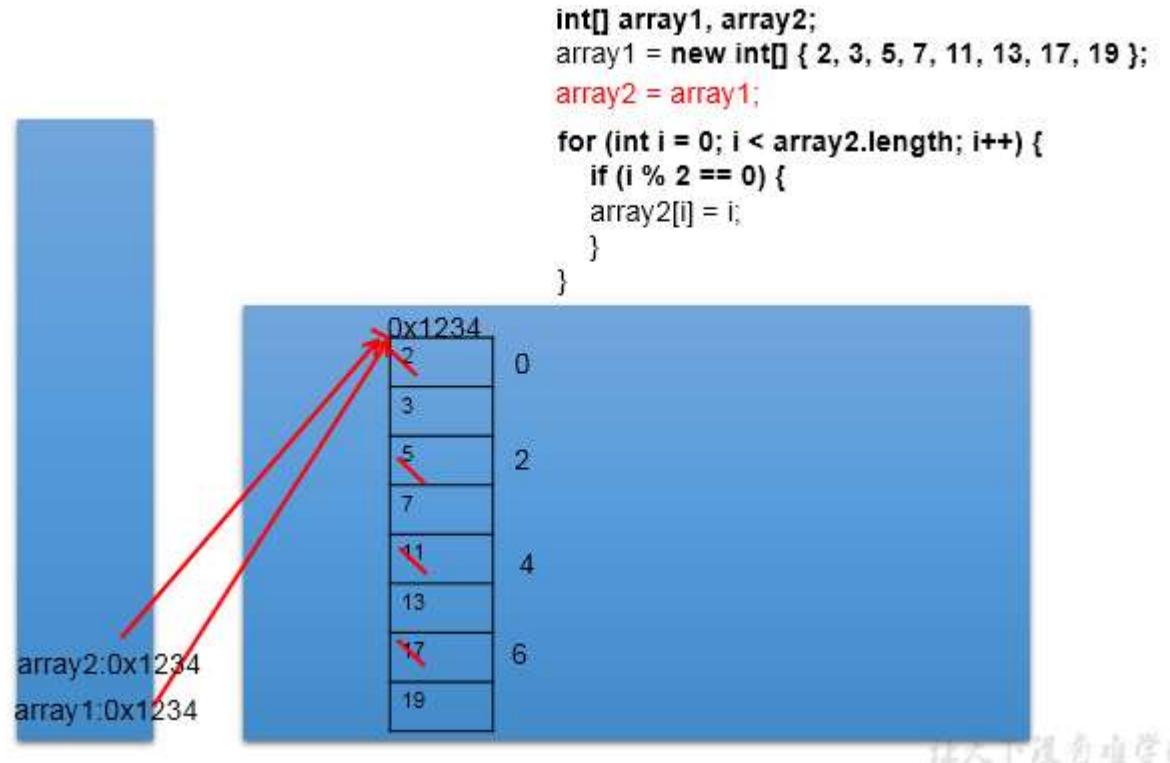
3. 数组的赋值与复制

```
int[] array1, array2;
array1 = new int[]{1,2,3,4};
```

3.1 赋值：

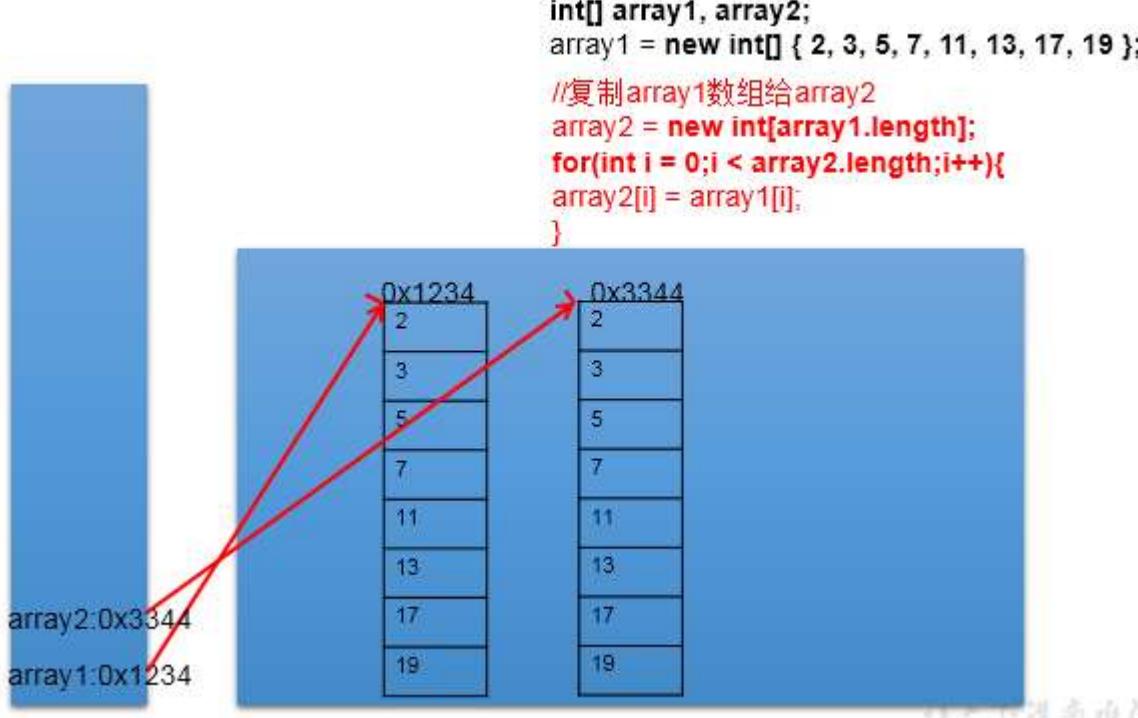
```
array2 = array1;
```

如何理解：将array1保存的数组的地址值赋给了array2，使得array1和array2共同指向堆空间中的同一个数组实体。



3.2 复制：

```
array2 = new int[array1.length];
for (int i = 0; i < array2.length; i++) {
    array2[i] = array1[i];
}
```



如何理解：我们通过new的方式，给array2在堆空间中新开辟了数组的空间。将array1数组中的元素值一个一个的赋值到array2数组中。

4. 数组元素的反转

```

//方法一:
//    for(int i = 0;i < arr.length / 2;i++){
//        String temp = arr[i];
//        arr[i] = arr[arr.length - i -1];
//        arr[arr.length - i -1] = temp;
//    }

//方法二:
//    for(int i = 0,j = arr.length - 1;i < j;i++,j--){
//        String temp = arr[i];
//        arr[i] = arr[j];
//        arr[j] = temp;
//    }

```

5. 数组中指定元素的查找：搜索、检索

5.1 线性查找：

实现思路：通过遍历的方式，一个一个的数据进行比较、查找。

适用性：具有普遍适用性。

5.2 二分法查找：

实现思路：每次比较中间值，折半的方式检索。

适用性：（前提：数组必须有序）

6. 数组的排序算法

十大内部排序算法

- 选择排序
 - 直接选择排序、**堆排序**
 - 交换排序
 - **冒泡排序、快速排序**
 - 插入排序
 - 直接插入排序、折半插入排序、**Shell排序**
 - 归并排序
 - 桶式排序
 - 基数排序
- 详细操作，见《附录：尚硅谷_宋红康_排序算法.pdf》

理解：

1) 衡量排序算法的优劣：

时间复杂度、空间复杂度、稳定性

2) 排序的分类：内部排序 与 外部排序（需要借助于磁盘）

3) 不同排序算法的时间复杂度

排序方法	时间复杂度（平均）	时间复杂度（最坏）	时间复杂度（最好）	空间复杂度	稳定性
插入排序	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	稳定
希尔排序	$O(n^{1.3})$	$O(n^2)$	$O(n)$	$O(1)$	不稳定
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定
堆排序	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(1)$	不稳定
冒泡排序	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	稳定
快速排序	$O(n \log_2 n)$	$O(n^2)$	$O(n \log_2 n)$	$O(n \log_2 n)$	不稳定
归并排序	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(n)$	稳定
计数排序	$O(n+k)$	$O(n+k)$	$O(n+k)$	$O(n+k)$	稳定
桶排序	$O(n+k)$	$O(n^2)$	$O(n)$	$O(n+k)$	稳定
基数排序	$O(n*k)$	$O(n*k)$	$O(n*k)$	$O(n+k)$	稳定

4) 手写冒泡排序

```

int[] arr = new int[]{43, 32, 76, -98, 0, 64, 33, -21, 32, 99};

//冒泡排序
for(int i = 0; i < arr.length - 1; i++){
    for(int j = 0; j < arr.length - 1 - i; j++){
        if(arr[j] > arr[j + 1]){
            int temp = arr[j];
            arr[j] = arr[j + 1];
            arr[j + 1] = temp;
        }
    }
}

```

Arrays工具类的使用

1. 理解：

- ① 定义在java.util包下。
- ② Arrays:提供了很多操作数组的方法。

2. 使用：

//1.boolean equals(int[] a,int[] b):判断两个数组是否相等。

```
int[] arr1 = new int[]{1,2,3,4};  
int[] arr2 = new int[]{1,3,2,4};  
boolean isEqual = Arrays.equals(arr1, arr2);  
System.out.println(isEqual);
```

//2.String toString(int[] a):输出数组信息。

```
System.out.println(Arrays.toString(arr1));
```

//3 void fill(int[] a,int val):将指定值填充到数组之中。

```
Arrays.fill(arr1,10);  
System.out.println(Arrays.toString(arr1));
```

//4 void sort(int[] a):对数组进行排序。

```
Arrays.sort(arr2);  
System.out.println(Arrays.toString(arr2));
```

//5 int binarySearch(int[] a,int key)

```
int[] arr3 = new int[]{-98,-34,2,34,54,66,79,105,210,333};  
int index = Arrays.binarySearch(arr3, 210);  
if(index >= 0){  
    System.out.println(index);  
}else{  
    System.out.println("未找到");  
}
```

数组的常见异常

1. 数组角标越界异常：ArrayIndexOutOfBoundsException

```
int[] arr = new int[]{1,2,3,4,5};

//      for(int i = 0;i <= arr.length;i++){
//          System.out.println(arr[i]);
//      }

//      System.out.println(arr[-2]);

//      System.out.println("hello");
2. 空指针异常：NullPointerException
    //情况一：
//      int[] arr1 = new int[]{1,2,3};
//      arr1 = null;
//      System.out.println(arr1[0]);

    //情况二：
//      int[][] arr2 = new int[4][];
//      System.out.println(arr2[0][0]);

    //情况：
String[] arr3 = new String[]{"AA","BB","CC"};
arr3[0] = null;
System.out.println(arr3[0].toString());
```

小知识：一旦程序出现异常，未处理时，就终止执行。

1. 面向对象学习的三条主线：

- * 1. Java类及类的成员：属性、方法、构造器；代码块、内部类
- *
- * 2. 面向对象的大特征：封装性、继承性、多态性、（抽象性）
- *
- * 3. 其它关键字：this、super、static、final、abstract、interface、package、import等
- *
- * “大处着眼，小处着手”

2. 面向对象与面向过程（理解）

1. 面向过程：强调的是功能行为，以函数为最小单位，考虑怎么做。
 2. 面向对象：强调具备了功能的对象，以类/对象为最小单位，考虑谁来做。

举例对比：人把大象装进冰箱。

3. 完成一个项目（或功能）的思路：

- 根据问题需要，选择问题所针对的**现实世界中的实体**。
- 从实体中寻找解决问题相关的属性和功能，这些属性和功能就形成了**概念世界中的类**。
- 把抽象的实体用计算机语言进行描述，**形成计算机世界中类的定义**。即借助某种程序语言，把类构造成计算机能够识别和处理的数据结构。
- 将**类实例化成计算机世界中的对象**。对象是计算机世界中解决问题的最终工具。

4. 面向对象中两个重要的概念：

- 类：对一类事物的描述，是抽象的、概念上的定义
 对象：是实际存在的该类事物的每个个体，因而也称为实例(instance)
 ➤ 面向对象程序设计的重点是类的设计
 ➤ 设计类，就是设计类的成员。

二者的关系：

对象，是由类new出来的，派生出来的。

5. 面向对象思想落地实现的规则一

- * 1. 创建类，设计类的成员
- * 2. 创建类的对象
- * 3. 通过“对象.属性”或“对象.方法”调用对象的结构

补充：几个概念的使用说明

- * 属性 = 成员变量 = field = 域、字段
- * 方法 = 成员方法 = 函数 = method
- * 创建类的对象 = 类的实例化 = 实例化类

6. 对象的创建与对象的内存解析

典型代码：

```
Person p1 = new Person();
```

```
Person p2 = new Person();
```

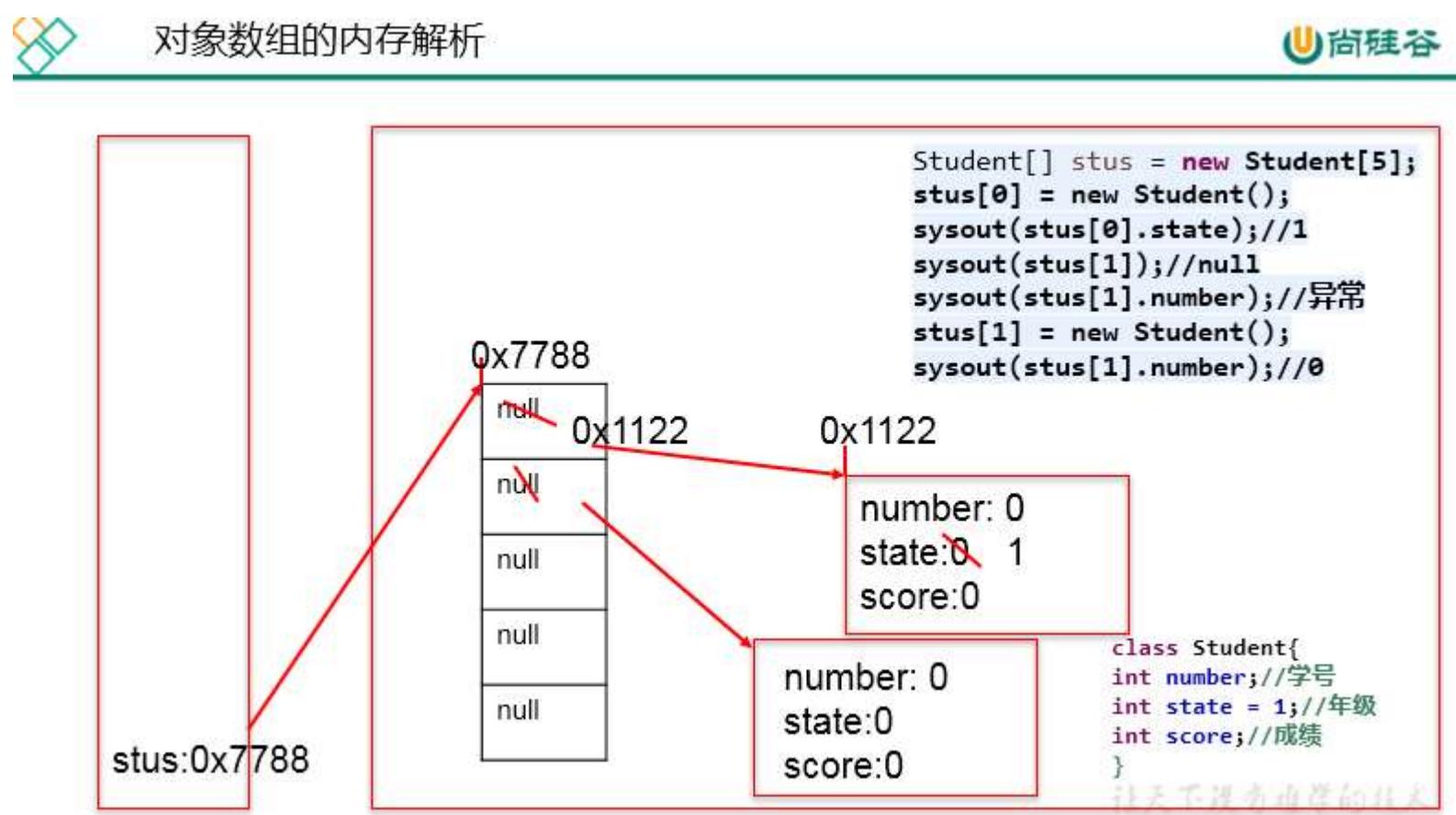
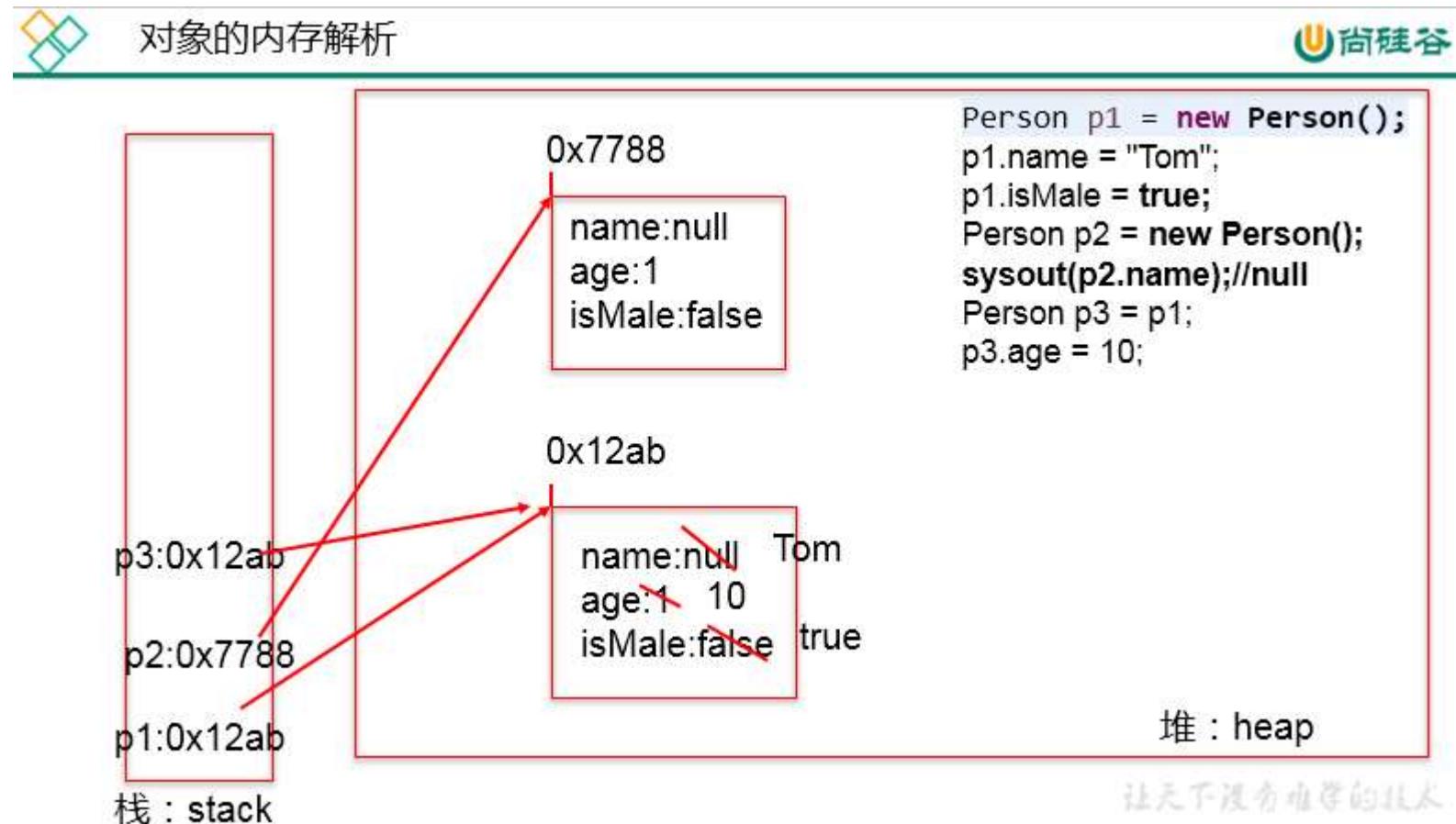
Person p3 = p1; //没有新创建一个对象，共用一个堆空间中的对象实体。

说明：

如果创建了一个类的多个对象，则每个对象都独立的拥有一套类的属性。（非 static 的）

意味着：如果我们修改一个对象的属性 a，则不影响另外一个对象属性 a 的值。

内存解析：



7. 匿名对象：我们创建的对象，没显式的赋给一个变量名。即为匿名对象

特点：匿名对象只能调用一次。

举例：

```
new Phone().sendEmail();
```

```
new Phone().playGame();

new Phone().price = 1999;
new Phone().showPrice() //0.0
```

应用场景：

```
PhoneMall mall = new PhoneMall();

//匿名对象的使用
mall.show(new Phone());
其中,
class PhoneMall{
    public void show(Phone phone){
        phone.sendEmail();
        phone.playGame();
    }
}
```

8. 理解"万事万物皆对象"

1. 在Java语言范畴中，我们都将功能、结构等封装到类中，通过类的实例化，来调用具体的功能结构

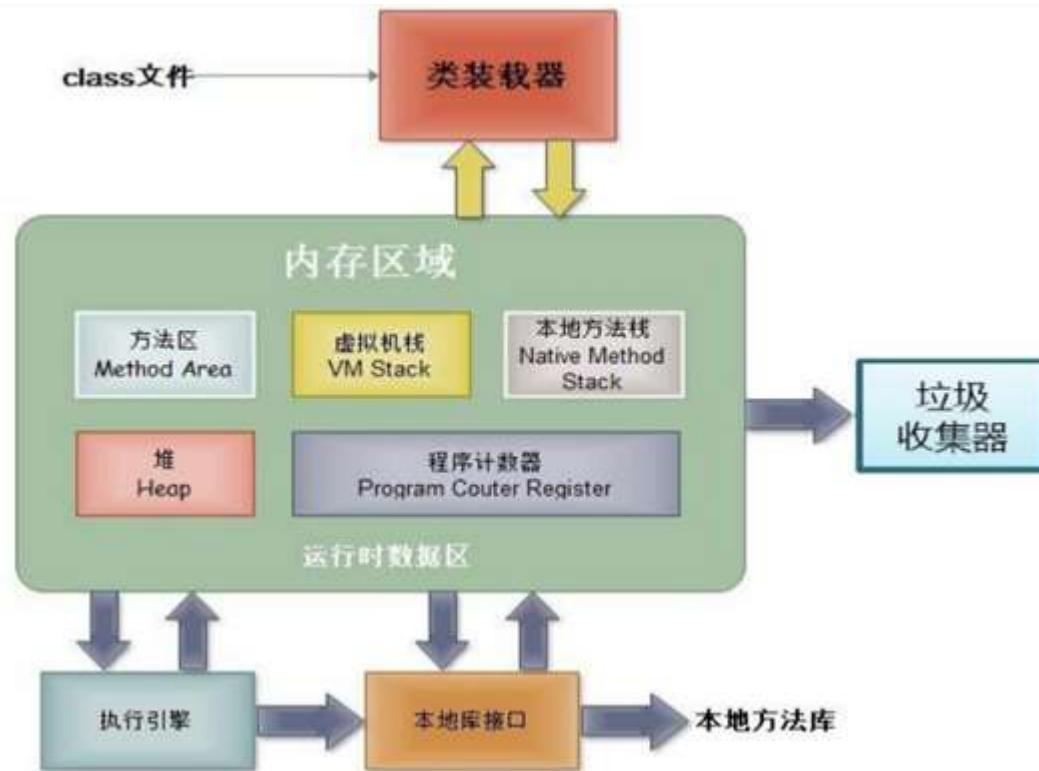
- * >Scanner, String等
- * >文件: File
- * >网络资源: URL

2. 涉及到Java语言与前端Html、后端的数据库交互时，前端的结构在Java层面交互时，都体现为类、对象。

JVM内存结构

编译完源程序以后，生成一个或多个字节码文件。

我们使用JVM中的类的加载器和解释器对生成的字节码文件进行解释运行。意味着，需要将字节码文件对应的类加载到内存中，涉及到内存解析。



《JVM规范》

虚拟机栈，即为平时提到的栈结构。**我们将局部变量存储在栈结构中**

堆，我们将new出来的结构（比如：数组、对象）加载在堆空间中。**补充：对象的属性（非static的）加载在堆空间中。**

方法区：类的加载信息、常量池、静态域

类的结构之一：属性

类的设计中，两个重要结构之一：属性

对比：属性 vs 局部变量

1. 相同点：

- * 1.1 定义变量的格式：数据类型 变量名 = 变量值
- * 1.2 先声明，后使用
- * 1.3 变量都其对应的作用域

2. 不同点：

2.1 在类中声明的位置的不同

属性：直接定义在类的一对{}内
局部变量：声明在方法内、方法形参、代码块内、构造器形参、构造器内部的

变量

2.2 关于权限修饰符的不同

属性：可以在声明属性时，指明其权限，使用权限修饰符。
常用的权限修饰符：private、public、缺省、protected ---->封装性
目前，大家声明属性时，都使用缺省就可以了。
局部变量：不可以使用权限修饰符。

2.3 默认初始化值的情况：

属性：类的属性，根据其类型，都默认初始化值。
整型（byte、short、int、long: 0）
浮点型（float、double: 0.0）
字符型（char: 0 （或'\u0000'））
布尔型（boolean: false）

引用数据类型（类、数组、接口：null）

局部变量：没默认初始化值。

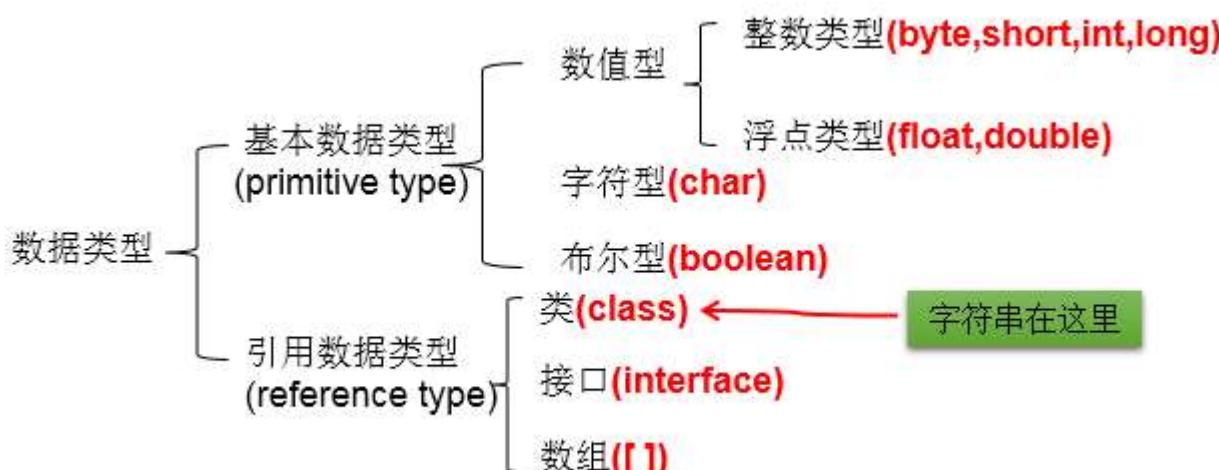
意味着，我们在调用局部变量之前，一定要显式赋值。
特别地：形参在调用时，我们赋值即可。

2.4 在内存中加载的位置：

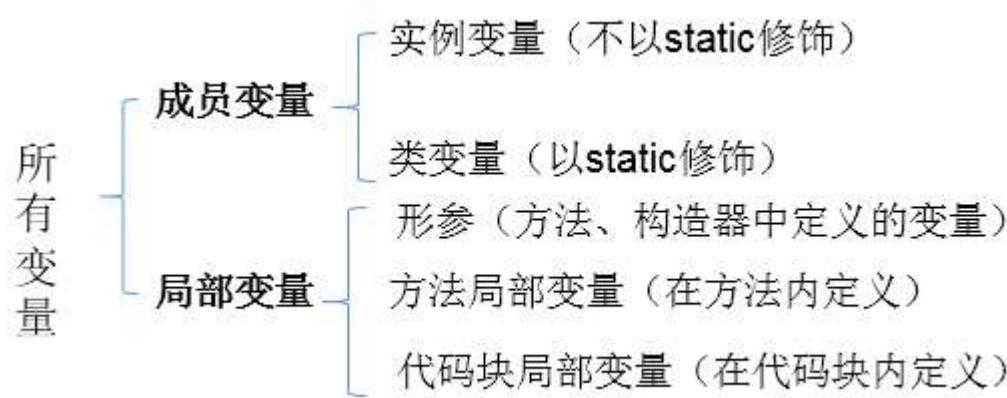
属性：加载到堆空间中（非static）
局部变量：加载到栈空间

补充：回顾变量的分类：

方式一：按照数据类型：



方式二：按照在类中声明的位置：



类的结构之二：方法

类的设计中，两个重要结构之二：方法

方法：描述类应该具有的功能。

```
* 比如：Math类：sqrt()\random() \...
* Scanner类：nextXxx() ...
* Arrays类：sort() \ binarySearch() \ toString() \ equals() \ ...
*
* 1. 举例：
* public void eat(){}
* public void sleep(int hour){}
* public String getName(){}
* public String getNation(String nation){}
*
* 2. 方法的声明：权限修饰符 返回值类型 方法名(形参列表){
*     方法体
* }
*
* 注意：static、final、abstract 来修饰的方法，后面再讲。
*
* 3. 说明：
*     3.1 关于权限修饰符：默认方法的权限修饰符先都使用public
*         Java规定的4种权限修饰符：private、public、缺省、protected -->封装性再细说
*
*     3.2 返回值类型：返回值 vs 没返回值
*         3.2.1 如果方法返回值，则必须在方法声明时，指定返回值的类型。同时，方法中，  
需要使用
*             return关键字来返回指定类型的变量或常量：“return 数据”。
*             如果方法没返回值，则方法声明时，使用void来表示。通常，没返回值的方法  
中，就不需要
*             使用return.但是，如果使用的话，只能“return;”表示结束此方法的意思。
*
*     3.2.2 我们定义方法该不该返回值？
*         ① 题目要求
*         ② 凭经验：具体问题具体分析
*
*     3.3 方法名：属于标识符，遵循标识符的规则和规范，“见名知意”
*
*     3.4 形参列表：方法可以声明0个，1个，或多个形参。
*         3.4.1 格式：数据类型1 形参1, 数据类型2 形参2, ...
*
*         3.4.2 我们定义方法时，该不该定义形参？
*             ① 题目要求
*             ② 凭经验：具体问题具体分析
*
*     3.5 方法体：方法功能的体现。
*
4. 方法的使用中，可以调用当前类的属性或方法
*     特殊的：方法A中又调用了方法A：递归方法。
*     方法中，不可以定义方法。
```

关键字: return

return关键字:

1. 使用范围: 使用在方法体中
2. 作用: ① 结束方法
* ② 针对于返回值类型的方法, 使用"return 数据"方法返回所要的数据。
3. 注意点: return关键字后面不可以声明执行语句。

1. 方法的重载的概念

定义：在同一个类中，允许存在一个以上的同名方法，只要它们的参数个数或者参数类型不同即可。

*

总结：“两同一不同”：同一个类、相同方法名

参数列表不同：参数个数不同，参数类型不同

2.

构成重载的举例：

举例一：Arrays类中重载的sort() / binarySearch(); PrintStream中的

println()

举例二：

//如下的4个方法构成了重载

```
public void getSum(int i,int j){
    System.out.println("1");
}

public void getSum(double d1,double d2){
    System.out.println("2");
}

public void getSum(String s ,int i){
    System.out.println("3");
}

public void getSum(int i,String s){
    System.out.println("4");
}
```

不构成重载的举例：

```
//如下的3个方法不能与上述4个方法构成重载
// public int getSum(int i,int j){
//     return 0;
// }

// public void getSum(int m,int n){
// }

// private void getSum(int i,int j){
// }
```

3. 如何判断是否构成方法的重载？

严格按照定义判断：两同一不同。

跟方法的权限修饰符、返回值类型、形参变量名、方法体都没关系！

4. 如何确定类中某一个方法的调用：

方法名 ---> 参数列表

面试题：方法的重载与重写的区别？

throws\throw
String\StringBuffer\StringBuilder
Collection\Collections
final\finally\finalize
...

抽象类、接口
sleep() / wait()

1. 使用说明：

- * 1. jdk 5.0新增的内容
- * 2. 具体使用：
 - * 2.1 可变个数形参的格式：数据类型 ... 变量名
 - * 2.2 当调用可变个数形参的方法时，传入的参数个数可以是：0个，1个,2个，。。。
 - * 2.3 可变个数形参的方法与本类中方法名相同，形参不同的方法之间构成重载
 - * 2.4 可变个数形参的方法与本类中方法名相同，形参类型也相同的数组之间不构成重载。换句话说，二者不能共存。
 - * 2.5 可变个数形参在方法的形参中，必须声明在末尾
 - * 2.6 可变个数形参在方法的形参中，最多只能声明一个可变形参。

2. 举例说明：

```
public void show(int i){  
}  
  
public void show(String s){  
    System.out.println("show(String)");  
}  
  
public void show(String ... strs){  
    System.out.println("show(String ... strs)");  
  
    for(int i = 0;i < strs.length;i++){  
        System.out.println(strs[i]);  
    }  
}  
//不能与上一个方法同时存在  
// public void show(String[] strs){  
//  
// }
```

调用时：

```
test.show("hello");  
test.show("hello","world");  
test.show();  
  
test.show(new String[]{"AA","BB","CC"});
```

1. 针对于方法内变量的赋值举例：

```

System.out.println("*****基本数据类型: *****");
int m = 10;
int n = m;

System.out.println("m = " + m + ", n = " + n);
n = 20;

System.out.println("m = " + m + ", n = " + n);

System.out.println("*****引用数据类型: *****");

Order o1 = new Order();
o1.orderId = 1001;

Order o2 = o1; //赋值以后，o1和o2的地址值相同，都指向了堆空间中同一个对象实体。

System.out.println("o1.orderId = " + o1.orderId + ", o2.orderId = " + o2.orderId);
o2.orderId = 1002;

System.out.println("o1.orderId = " + o1.orderId + ", o2.orderId = " + o2.orderId);

```

规则：

如果变量是基本数据类型，此时赋值的是变量所保存的数据值。

如果变量是引用数据类型，此时赋值的是变量所保存的数据的地址值。

2. 针对于方法的参数概念

形参：方法定义时，声明的小括号内的参数

实参：方法调用时，实际传递给形参的数据

3. java中参数传递机制：值传递

规则：

- * 如果参数是基本数据类型，此时实参赋给形参的是实参真实存储的数据值。
- * 如果参数是引用数据类型，此时实参赋给形参的是实参存储数据的地址值。

推广：

如果变量是基本数据类型，此时赋值的是变量所保存的数据值。

如果变量是引用数据类型，此时赋值的是变量所保存的数据的地址值。

4. 典型例题与内存解析：

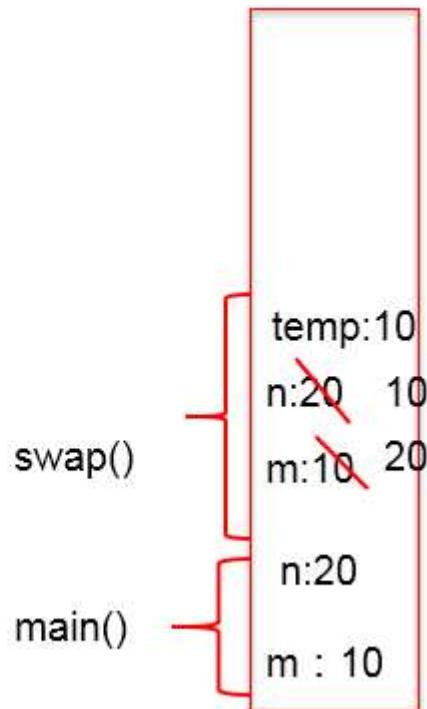
【例题1】

```

main(){
    int m = 10;
    int n = 20;
    v.swap(m,n);
    sysout(m,n);
}

swap(int m ,in n){
    int temp = m;
    m = n;
    n = temp;
    sysout(m,n);
}

```



【例题2】

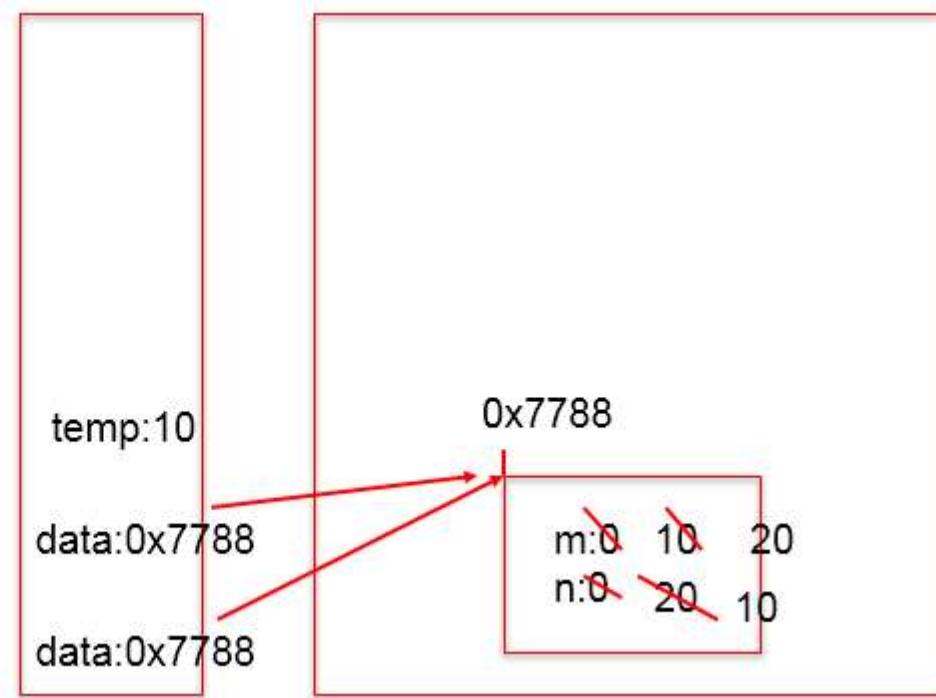
```

class Data{
    int m;
    int n;
}

main(){
    Data data = new Data();
    data.m = 10;
    data.n = 20;
    v.swap(data);
    sysout(data.m,data.n);
}

swap(Data data){
    int temp = data.m;
    data.m = data.n;
    data.n = temp;
}

```



1. 定义：

递归方法：一个方法体内调用它自身。

2. 如何理解递归方法？

> 方法递归包含了一种隐式的循环，它会重复执行某段代码，但这种重复执行无须循环控制。

> 递归一定要向已知方向递归，否则这种递归就变成了无穷递归，类似于死循环。

3. 举例：

```
// 例1：计算1-n之间所有自然数的和
public int getSum(int n) { // 3
```

```
    if (n == 1) {
        return 1;
    } else {
        return n + getSum(n - 1);
    }
```

```
}
```

// 例2：计算1-n之间所有自然数的乘积:n!

```
public int getSum1(int n) {

    if (n == 1) {
        return 1;
    } else {
        return n * getSum1(n - 1);
    }
```

```
}
```

//例3：已知一个数列：f(0) = 1, f(1) = 4, f(n+2)=2*f(n+1) + f(n)，
//其中n是大于0的整数，求f(10)的值。

```
public int f(int n){
    if(n == 0){
        return 1;
    }else if(n == 1){
        return 4;
    }else{
        //return f(n + 2) - 2 * f(n + 1);
        return 2*f(n - 1) + f(n - 2);
    }
}
```

//例4：斐波那契数列

//例5：汉诺塔问题

//例6：快排

面向对象的特征一：封装性

面向对象的特征一：封装与隐藏

1. 为什么要引入封装性？

- 我们程序设计追求“高内聚，低耦合”。
 - ① 高内聚：类的内部数据操作细节自己完成，不允许外部干涉；
 - ② 低耦合：仅对外暴露少量的方法用于使用。
- 隐藏对象内部的复杂性，只对外公开简单的接口。便于外界调用，从而提高系统的可扩展性、可维护性。通俗的说，**把该隐藏的隐藏起来，该暴露的暴露出来。这就是封装性的设计思想。**

2. 问题引入：

当我们创建一个类的对象以后，我们可以通过“对象.属性”的方式，对对象的属性进行赋值。这里，赋值操作要受到属性的数据类型和存储范围的制约。除此之外，没其他制约条件。但是，在实际问题中，我们往往需要给属性赋值加入额外的限制条件。这个条件就不能在属性声明时体现，我们只能通过方法进行限制条件的添加。（比如：`setLegs()`同时，我们需要避免用户再使用“对象.属性”的方式对属性进行赋值。则需要将属性声明为私有的（`private`）。

-->此时，针对于属性就体现了封装性。

3. 封装性思想具体的代码体现：

体现一：将类的属性`xxx`私化(`private`)，同时，提供公共的(`public`)方法来获取(`getXXX`)和设置(`setXXX`)此属性的值

```
private double radius;
public void setRadius(double radius){
    this.radius = radius;
}
public double getRadius(){
    return radius;
}
```

体现二：不对外暴露的私有的方法

体现三：单例模式（将构造器私有化）

体现四：如果不希望类在包外被调用，可以将类设置为缺省的。

4. Java规定的四种权限修饰符

4.1 权限从小到大顺序为： `private < 缺省 < protected < public`

4.2 具体的修饰范围：

修饰符	类内部	同一个包	不同包的子类	同一个工程
<code>private</code>	Yes			
(缺省)	Yes	Yes		
<code>protected</code>	Yes	Yes	Yes	
<code>public</code>	Yes	Yes	Yes	Yes

4.3 权限修饰符可用来修饰的结构说明：

**4种权限都可以用来修饰类的内部结构：属性、方法、构造器、内部类
修饰类的话，只能使用：缺省、`public`**

1. 构造器（或构造方法）：Constructor

构造器的作用：

- * 1. 创建对象
- * 2. 初始化对象的信息

2. 使用说明：

- * 1. 如果没显式的定义类的构造器的话，则系统默认提供一个空参的构造器
- * 2. 定义构造器的格式：权限修饰符 类名(形参列表){}
- * 3. 一个类中定义的多个构造器，彼此构成重载
- * 4. 一旦我们显式的定义了类的构造器之后，系统就不再提供默认的空参构造器
- * 5. 一个类中，至少会有一个构造器。

3. 举例：

```
//构造器
public Person(){
    System.out.println("Person().....");
}

public Person(String n){
    name = n;
}

public Person(String n,int a){
    name = n;
    age = a;
}
```

属性赋值顺序

- * 总结：属性赋值的先后顺序
- *
- *
- * ① 默认初始化
- * ② 显式初始化
- * ③ 构造器中初始化
- * *****
- * ④ 通过"对象.方法" 或 "对象.属性"的方式，赋值
- *
- * 以上操作的先后顺序： ① - ② - ③ - ④

JavaBean的概念

所谓**JavaBean**, 是指符合如下标准的**Java**类:

- >类是公共的
- >一个无参的公共的构造器
- >属性, 且对应的**get**、**set**方法

关键字: this

1. 可以调用的结构: 属性、方法; 构造器

2. this调用属性、方法:

this理解为: 当前对象 或 当前正在创建的对象

2.1 在类的方法中, 我们可以使用"this.属性"或"this.方法"的方式, 调用当前对象属性或方法。但是,

* 通常情况下, 我们都择省略"this.". 特殊情况下, 如果方法的形参和类的属性同名时, 我们必须显式
* 的使用"this.变量"的方式, 表明此变量是属性, 而非形参。
*

* 2.2 在类的构造器中, 我们可以使用"this.属性"或"this.方法"的方式, 调用当前正在创建的对象属性或方法。但是, 通常情况下, 我们都择省略"this.". 特殊情况下, 如果构造器的形参和类的属性同名时, 我们必须显式的使用"this.变量"的方式, 表明此变量是属性, 而非形参。

3. this调用构造器:

- ① 我们在类的构造器中, 可以显式的使用"this(形参列表)"方式, 调用本类中指定的其他构造器
- ② 构造器中不能通过"this(形参列表)"方式调用自己
- ③ 如果一个类中有n个构造器, 则最多有 n - 1构造器中使用了"this(形参列表)"
- ④ 规定: "this(形参列表)"必须声明在当前构造器的首行
- ⑤ 构造器内部, 最多只能声明一个"this(形参列表)", 用来调用其他的构造器

1. package的使用

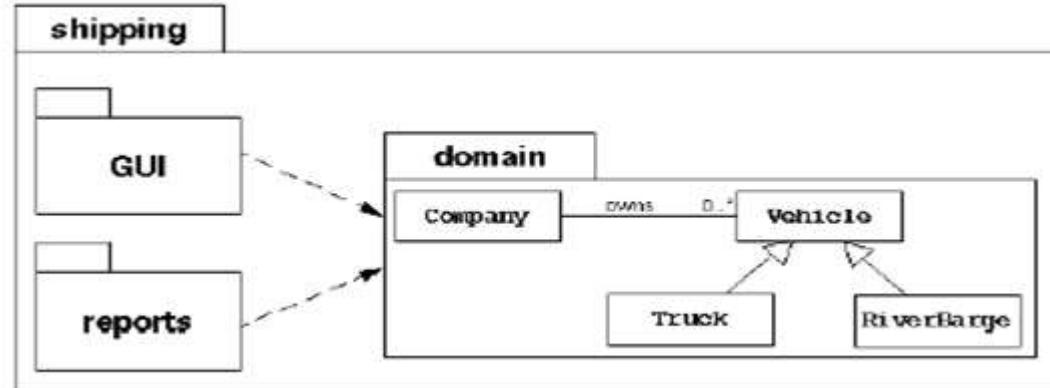
1.1 使用说明:

- * 1.为了更好的实现项目中类的管理，提供包的概念
- * 2.使用package声明类或接口所属的包，声明在源文件的首行
- * 3.包，属于标识符，遵循标识符的命名规则、规范(xxxxxxxx)、“见名知意”
- * 4.每"."一次，就代表一层文件目录。

1.2 举例:

举例一：

某航运软件系统包括：一组域对象、GUI和reports子系统



举例二：MVC设计模式

模型层 model 主要处理数据

- >数据对象封装 model.bean/domain
- >数据库操作类 model.dao
- >数据库 model.db

控制层 controller 处理业务逻辑

- >应用界面相关 controller.activity
- >存放fragment controller.fragment
- >显示列表的适配器 controller.adapter
- >服务相关的 controller.service
- >抽取的基类 controller.base

视图层 view 显示数据

- >相关工具类 view.utils
- >自定义view view.ui

1.3 JDK中的主要包介绍:

1. **java.lang**----包含一些Java语言的核心类，如String、Math、Integer、System和Thread，提供常用功能
2. **java.net**----包含执行与网络相关的操作的类和接口。
3. **java.io** ----包含能提供多种输入/输出功能的类。
4. **java.util**----包含一些实用工具类，如定义系统特性、接口的集合框架类、使用与日期日历相关的函数。
5. **java.text**----包含了一些java格式化相关的类
6. **java.sql**----包含了java进行JDBC数据库编程的相关类/接口
7. **java.awt**----包含了构成抽象窗口工具集 (abstract window toolkits) 的多个类，这些类被用来构建和管理应用程序的图形用户界面(GUI)。 B/S C/S

2. import的使用:

import:导入

- * 1. 在源文件中显式的使用import结构导入指定包下的类、接口
- * 2. 声明在包的声明和类的声明之间
- * 3. 如果需要导入多个结构，则并列写出即可
- * 4. 可以使用"xxx.*"的方式，表示可以导入xxx包下的所有结构
- * 5. 如果使用的类或接口是java.lang包下定义的，则可以省略import结构

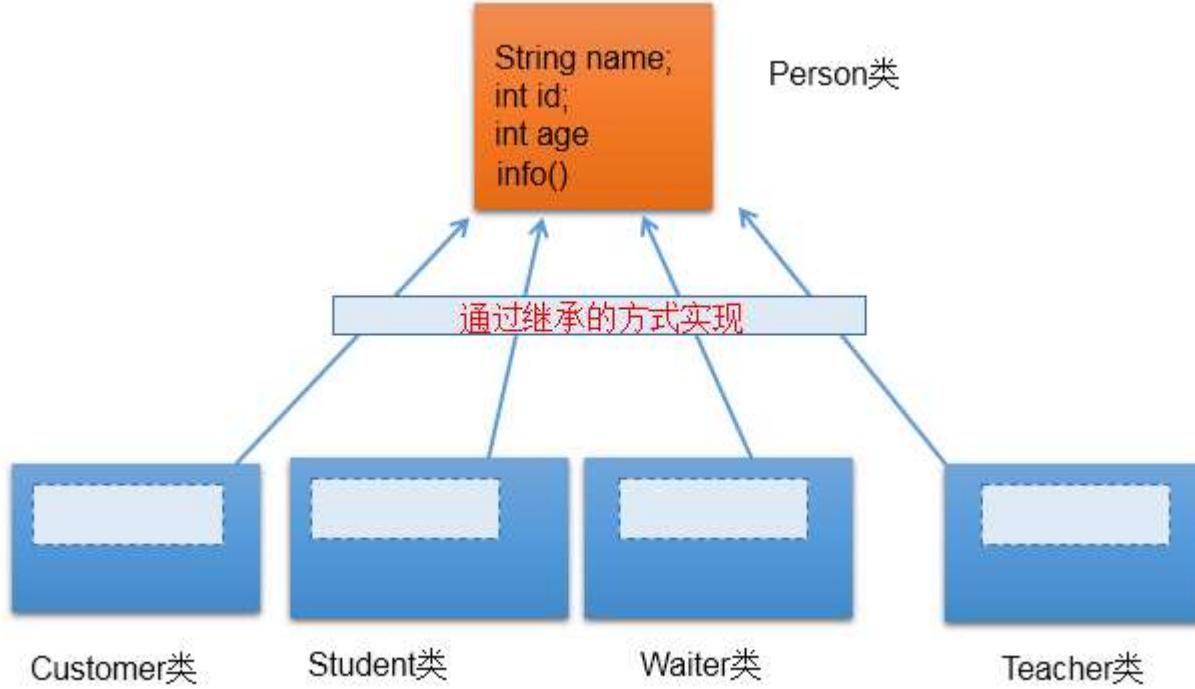
- * 6. 如果使用的类或接口是本包下定义的，则可以省略**import**结构
- * 7. 如果在源文件中，使用了不同包下的同名的类，则必须至少一个类需要以全类名的方式显示。
- * 8. 使用“**xxx.***”方式表明可以调用**xxx**包下的所有结构。但是如果使用的是**xxx**子包下的结构，则仍需要显式导入
- *
- * 9. **import static**: 导入指定类或接口中的静态结构:属性或方法。

面向对象的特征二：继承性

1.为什么要有类的继承性？（继承性的好处）

- * ① 减少了代码的冗余，提高了代码的复用性
- * ② 便于功能的扩展
- * ③ 为之后多态性的使用，提供了前提

图示：



2.继承性的格式：

```
class A extends B{}  
*   A:子类、派生类、subclass  
*   B:父类、超类、基类、superclass
```

3.子类继承父类以后有哪些不同？

3.1体现：一旦子类A继承父类B以后，子类A中就获取了父类B中声明的**所有的属性和方法**。

* 特别的，父类中声明为**private**的属性或方法，子类继承父类以后，仍然认为获取了父类中私的结构。只因为封装性的影响，使得子类不能直接调用父类的结构而已。

3.2 子类继承父类以后，还可以声明自己特有的属性或方法：实现功能的拓展。

- * 子类和父类的关系，不同于子集和集合的关系。
- * **extends:** 延展、扩展

4. Java中继承性的说明

1.一个类可以被多个子类继承。

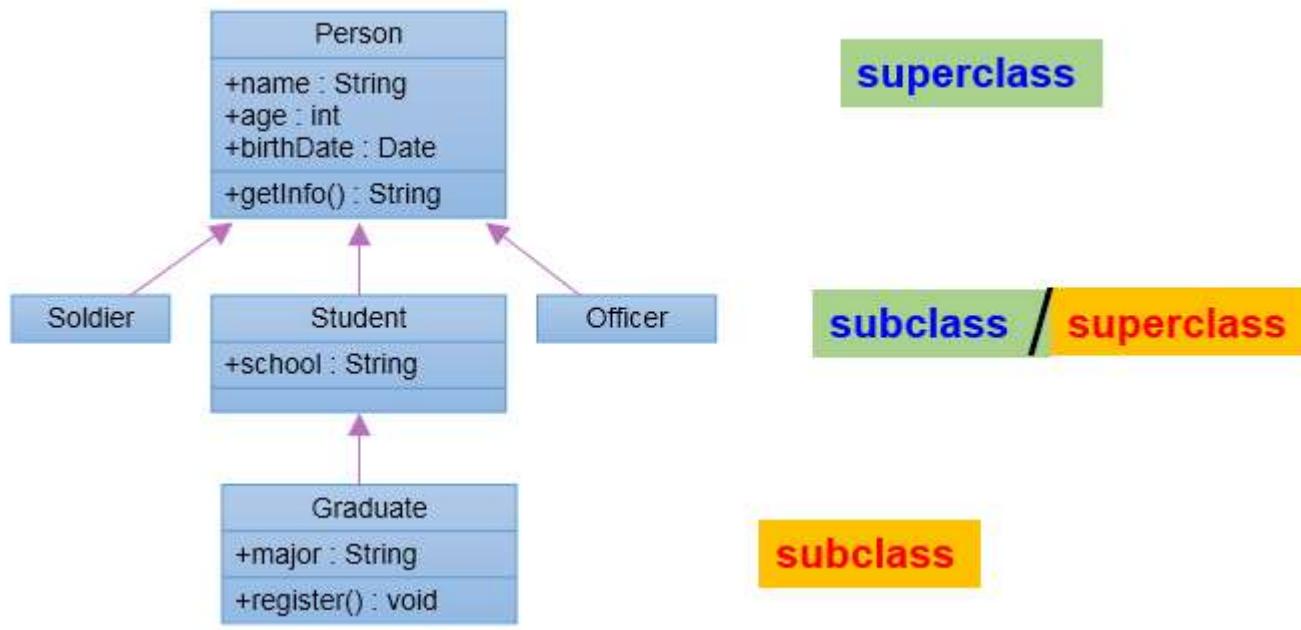
2. Java中类的单继承性：一个类只能有一个父类

3.子父类是相对的概念。

4.子类直接继承的父类，称为：直接父类。间接继承的父类称为：间接父类

5.子类继承父类以后，就获取了直接父类以及所间接父类中声明的属性和方法

图示：



5. java.lang.Object类的理解

1. 如果我们没显式的声明一个类的父类的话，则此类继承于java.lang.Object类
2. 所的java类（除java.lang.Object类之外都直接或间接的继承于java.lang.Object类
3. 意味着，所的java类具有java.lang.Object类声明的功能。

方法的重写

1. 什么是方法的重写(override 或 overwrite)?

子类继承父类以后，可以对父类中同名同参数的方法，进行覆盖操作。

2. 应用：

重写以后，当创建子类对象以后，通过子类对象调用子父类中的同名同参数的方法时，实际执行的是子类重写父类的方法。

3. 举例：

```
class Circle{
    public double findArea(){}/>求面积
}

class Cylinder extends Circle{
    public double findArea(){}/>求表面积
}

*****
class Account{
    public boolean withdraw(double amt){}
}

class CheckAccount extends Account{
    public boolean withdraw(double amt){}
}
```

4. 重写的规则：

方法的声明： 权限修饰符 返回值类型 方法名(形参列表) throws 异常的类型{
 * //方法体
 * }
 * 约定俗称： 子类中的叫重写的方法，父类中的叫被重写的方法
 * ① 子类重写的方法的方法名和形参列表与父类被重写的方法的方法名和形参列表相同
 * ② 子类重写的方法的权限修饰符不小于父类被重写的方法的权限修饰符
 * >特殊情况：子类不能重写父类中声明为private权限的方法
 * ③ 返回值类型：
 * >父类被重写的方法的返回值类型是void，则子类重写的方法的返回值类型只能是void
 * >父类被重写的方法的返回值类型是A类型，则子类重写的方法的返回值类型可以是A类或A类的子类
 * >父类被重写的方法的返回值类型是基本数据类型(比如：double)，则子类重写的方法的返回值类型必须是相同的基本数据类型(必须也是double)
 * ④ 子类重写的方法抛出的异常类型不大于父类被重写的方法抛出的异常类型 (具体放到异常处理时候讲)
 * ****
 * 子类和父类中的同名同参数的方法要么都声明为非static的 (考虑重写，要么都声明为static的 (不是重写))。

5. 面试题：

区分方法的重写和重载？

答：

- ① 二者概念：
- ② 重载和重写的具体规则
- ③ 重载：不表现为多态性。

重写：表现为多态性。

重载，是指允许存在多个同名方法，而这些方法的参数不同。编译器根据方法不同的参数表，对同名方法的名称做修饰。对于编译器而言，这些同名方法就成了不同的方法。**它们的调用地址在编译期就绑定了**。Java的重载是可以包括父类和子类的，即子类可以重载父类的同名不同参数的方法。

所以：对于重载而言，在方法调用之前，编译器就已经确定了所要调用的方法，这称为**“早绑定”或“静态绑定”**；

而对于多态，只等到方法调用的那一刻，解释运行器才会确定所要调用的具体方法，这称为**“晚绑定”或“动态绑定”**。

引用一句Bruce Eckel的话：“**不要犯傻，如果它不是晚绑定，它就不是多态。**”

关键字: super

1. **super** 关键字可以理解为: 父类的

2. 可以用来调用的结构:

属性、方法、构造器

3. **super** 调用属性、方法:

3.1 我们可以在子类的方法或构造器中。通过使用"**super.属性**"或"**super.方法**"的方式，显式的调用父类中声明的属性或方法。但是，通常情况下，我们习惯省略"**super.**"

3.2 特殊情况: 当**子类和父类中定义了同名的属性**时，我们要想在子类中调用父类中声明的属性，则必须显式的使用"**super.属性**"的方式，表明调用的是父类中声明的属性。

3.3 特殊情况: 当**子类重写了父类中的方法**以后，我们想在子类的方法中调用父类中被重写的方法时，则必须显式的使用"**super.方法**"的方式，表明调用的是父类中被重写的方法。

4. **super** 调用构造器:

4.1 我们可以在子类的构造器中显式的使用"**super(形参列表)**"的方式，调用父类中声明的指定的构造器

4.2 "**super(形参列表)**"的使用，必须声明在子类构造器的首行！

4.3 我们在类的构造器中，针对于"**this(形参列表)**"或"**super(形参列表)**"只能二一，不能同时出现

4.4 在构造器的首行，没显式的声明"**this(形参列表)**"或"**super(形参列表)**"，则默认调用的是父类中空参的构造器: **super()**

4.5 在类的多个构造器中，至少一个类的构造器中使用了"**super(形参列表)**"，调用父类中的构造器

理解即可。

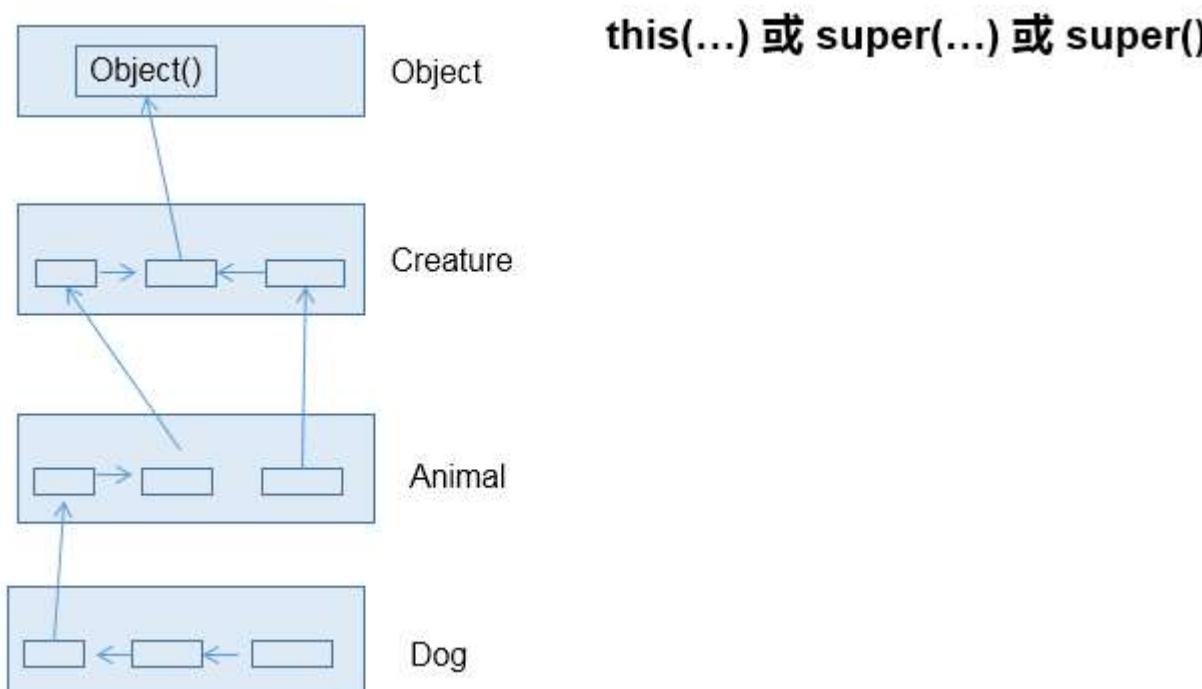
1. 从结果上看：继承性

- > 子类继承父类以后，就获取了父类中声明的属性或方法。
- > 创建子类的对象，在堆空间中，就会加载所父类中声明的属性。

2. 从过程上看：

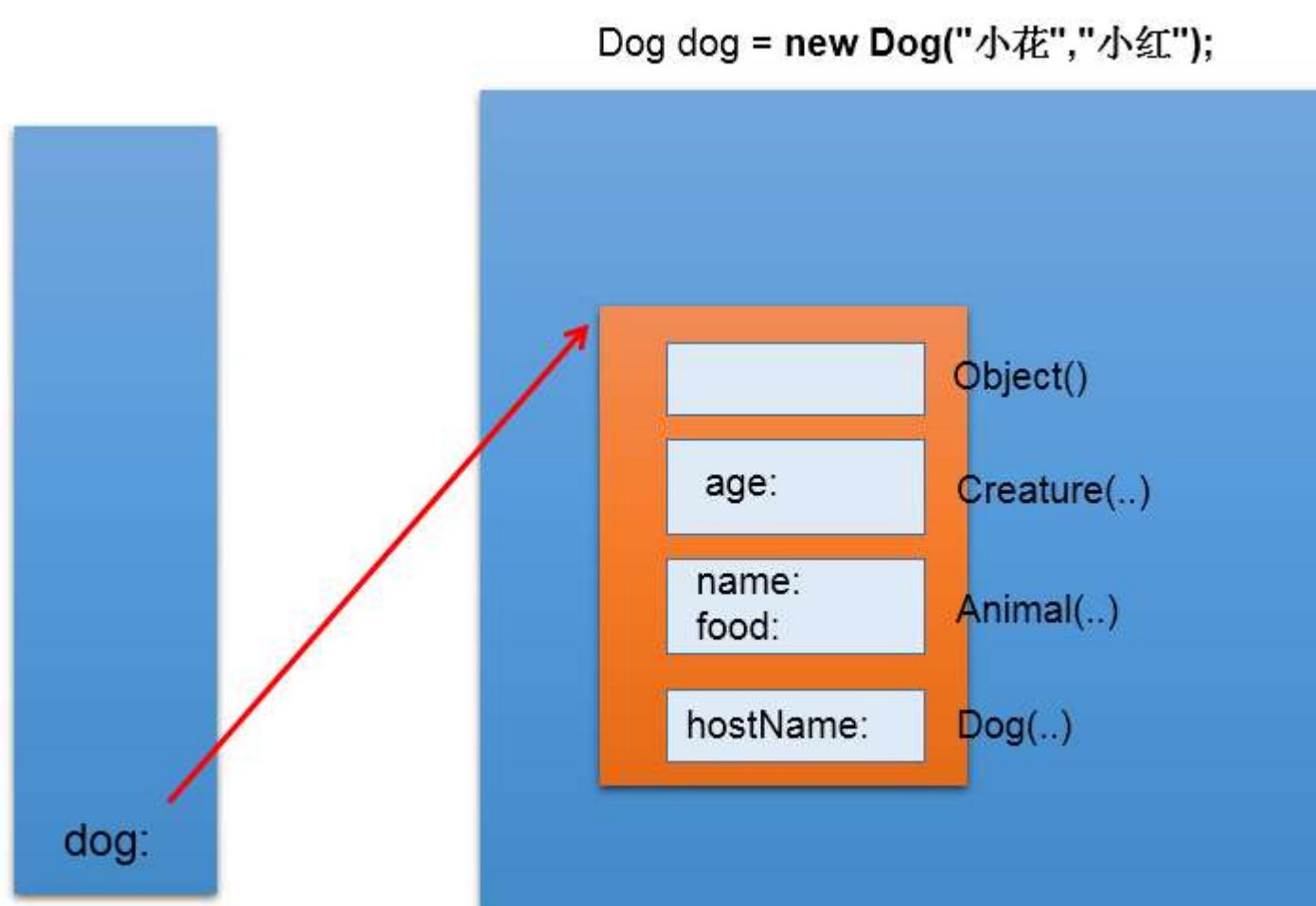
当我们通过子类的构造器创建子类对象时，我们一定会直接或间接的调用其父类的构造器，进而调用父类的父类的构造器，...直到调用了`java.lang.Object`类中空参的构造器为止。正因为加载过所的父类的结构，所以才可以看到内存中父类中的结构，子类对象才可以考虑进行调用。

图示：



3. 强调说明：

虽然创建子类对象时，调用了父类的构造器，但是自始至终就创建过一个对象，即为`new`的子类对象。



1. 多态性的理解：可以理解为一个事物的多种形态。

2. 何为多态性：

对象的多态性：父类的引用指向子类的对象（或子类的对象赋给父类的引用）
举例：

```
Person p = new Man();
```

```
Object obj = new Date();
```

3. 多态性的使用：虚拟方法调用

- > 有了对象的多态性以后，我们在编译期，只能调用父类中声明的方法，但在运行期，我们实际执行的是子类重写父类的方法。
- > 总结：编译，看左边；运行，看右边。

4. 多态性的使用前提：

① 类的继承关系 ② 方法的重写

5. 多态性的应用举例：

举例一：

```
public void func(Animal animal){ //Animal animal = new Dog();
    animal.eat();
    animal.shout();
}
```

举例二：

```
public void method(Object obj){
}
```

举例三：

```
class Driver{

    public void doData(Connection conn){ //conn = new MySQLConnection(); / conn = new
    OracleConnection();
        // 规范的步骤去操作数据
        // conn.method1();
        // conn.method2();
        // conn.method3();

    }
}
```

6. 多态性使用的注意点：

对象的多态性，只适用于方法，不适用于属性（编译和运行都看左边）

7. 关于向上转型与向下转型：

7.1 向上转型：多态

7.2 向下转型：

7.2.1 为什么使用向下转型：

有了对象的多态性以后，内存中实际上是加载了子类特有的属性和方法的，但是由于变量声明为父类类型，导致编译时，只能调用父类中声明的属性和方法。子类特有的属性和方法不能调用。如何才能调用子类特有的属性和方法？使用向下转型。

7.2.2 如何实现向下转型：

使用强制类型转换符：()

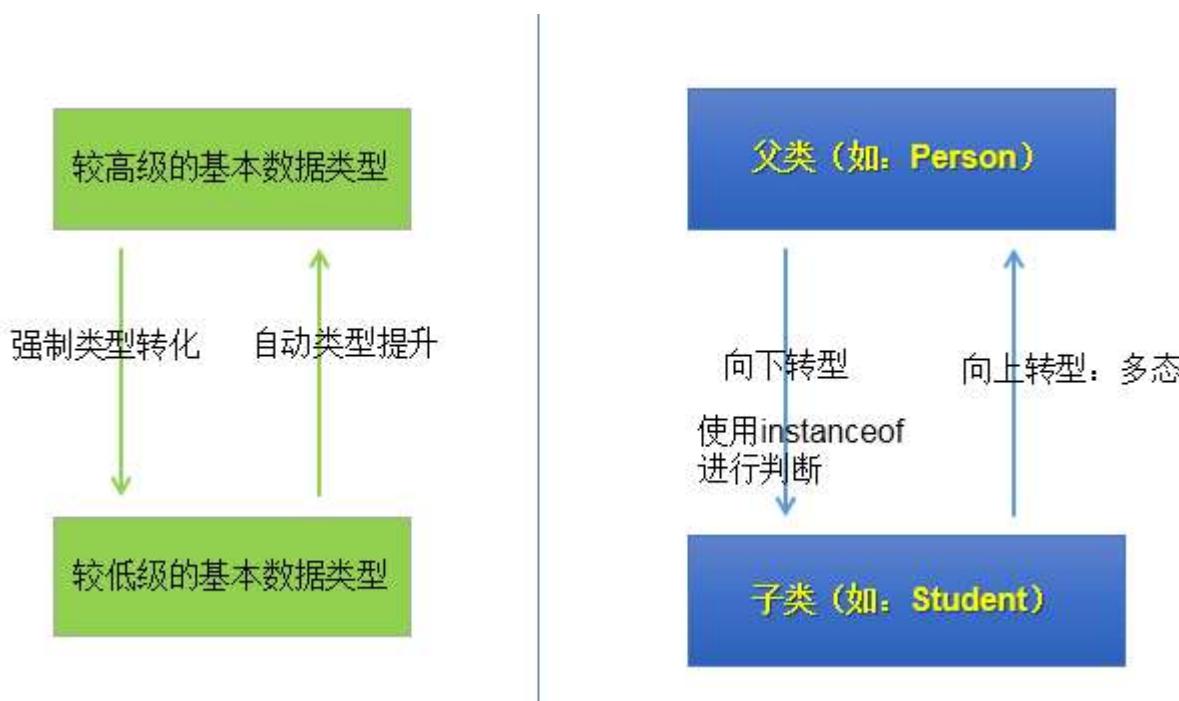
7.2.3 使用时的注意点：

- ① 使用强转时，可能出现ClassCastException的异常。
- ② 为了避免在向下转型时出现ClassCastException的异常，我们在向下转型之前，先进行`instanceof`的判断，一旦返回true，就进行向下转型。如果返回false，不进行向下转型。

7.2.4 instanceof的使用：

- ① `a instanceof A`: 判断对象a是否是类A的实例。如果是，返回true；如果不是，返回false。
- ② 如果 `a instanceof A` 返回true，则 `a instanceof B` 也返回true. 其中，类B是类A的父类。
- ③ 要求a所属的类与类A必须是子类和父类的关系，否则编译错误。

7.2.5 图示：



8. 面试题：

8.1 谈谈你对多态性的理解？

- ① 实现代码的通用性。
- ② Object类中定义的public boolean equals(Object obj){ }
- JDBC: 使用java程序操作(获取数据库连接、CRUD)数据库(MySQL、Oracle、DB2、SQL Server)
- ③ 抽象类、接口的使用肯定体现了多态性。（抽象类、接口不能实例化）

8.2 多态是编译时行为还是运行时行为？

1.java.lang.Object类的说明：

- * 1.Object类是所有Java类的根父类
- * 2.如果在类的声明中未使用extends关键字指明其父类，则默认父类为java.lang.Object类
- * 3.Object类中的功能(属性、方法)就具通用性。
- * 属性：无
- * 方法：equals() / toString() / getClass() / hashCode() / clone() / finalize()
- * wait()、notify()、notifyAll()
- *
- * 4. Object类只声明了一个空参的构造器

2.equals()方法

2.1 equals()的使用：

1. 是一个方法，而非运算符
- * 2. 只能适用于引用数据类型
- * 3. Object类中equals()的定义：


```
public boolean equals(Object obj) {
    return (this == obj);
```
- * 说明：Object类中定义的equals()和==的作用是相同的：比较两个对象的地址值是否相同。即两个引用是否指向同一个对象实体
- *
- * 4. 像String、Date、File、包装类等都重写了Object类中的equals()方法。重写以后，比较的不是两个引用的地址是否相同，而是比较两个对象的“实体内容”是否相同。
- *
- * 5. 通常情况下，我们自定义的类如果使用equals()的话，也通常是比较两个对象的“实体内容”是否相同。那么，我们
 - * 就需要对Object类中的equals()进行重写。
 - * 重写的原则：比较两个对象的实体内容是否相同。

2.2 如何重写equals()

2.2.1 手动重写举例：

```
class User{
    String name;
    int age;
    //重写其equals()方法
    public boolean equals(Object obj){
        if(obj == this){
            return true;
        }
        if(obj instanceof User){
            User u = (User)obj;
            return this.age == u.age && this.name.equals(u.name);
        }
        return false;
    }
}
```

2.2.2 开发中如何实现：自动生成的

2.3 回顾 == 运算符的使用：

* == : 运算符
 * 1. 可以使用在基本数据类型变量和引用数据类型变量中
 * 2. 如果比较的是基本数据类型变量：比较两个变量保存的数据是否相等。（不一定类型要相同）
 * 如果比较的是引用数据类型变量：比较两个对象的地址值是否相同，即两个引用是否指向同一个对象实体
 * 补充： == 符号使用时，必须保证符号左右两边的变量类型一致。

3. `toString()`方法

3.1 `toString()`的使用：

1. 当我们输出一个对象的引用时，实际上就是调用当前对象的`toString()`
 *
 * 2. Object类中`toString()`的定义：
 * public String `toString()` {
 return getClass().getName() + "@" + Integer.toHexString(hashCode());
 }
 *
 * 3. 像String、Date、File、包装类等都重写了Object类中的`toString()`方法。
 * 使得在调用对象的`toString()`时，返回"实体内容"信息
 *
 * 4. 自定义类也可以重写`toString()`方法，当调用此方法时，返回对象的"实体内容"

3.2 如何重写`toString()`

举例：

```
//自动实现
@Override
public String toString() {
    return "Customer [name=" + name + ", age=" + age + "]";
}
```

4. 面试题：

- ① final、finally、finalize的区别？
- ② == 和 equals() 区别

单元测试方法

```
* Java中的JUnit单元测试
*
* 步骤：
* 1. 在当前工程 - 右键择: build path - add libraries - JUnit 4 - 下一步
* 2. 创建Java类，进行单元测试。
* 此时的Java类要求：① 此类是public的 ②此类提供公共的无参的构造器
* 3. 此类中声明单元测试方法。
* 此时的单元测试方法：方法的权限是public, 没返回值，没形参
*
* 4. 此单元测试方法上需要声明注解：@Test，并在单元测试类中导入：import
org.junit.Test;
*
* 5. 声明好单元测试方法以后，就可以在方法体内测试相关的代码。
* 6. 写完代码以后，左键双击单元测试方法名，右键：run as - JUnit Test
*
* 说明：
* 1. 如果执行结果没任何异常：绿条
* 2. 如果执行结果出现异常：红条
```

包装类的使用

1.为什么要有包装类(或封装类)

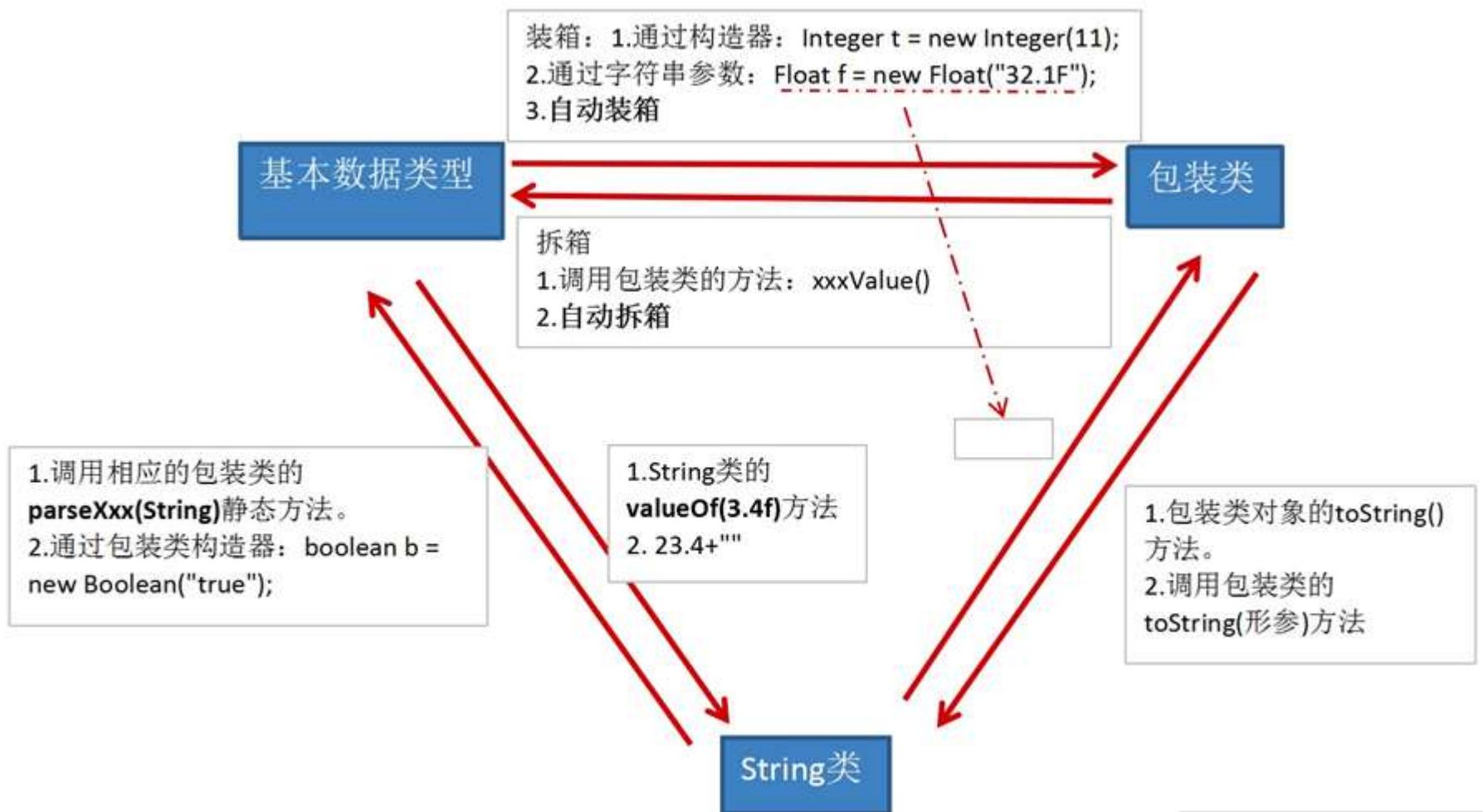
为了使基本数据类型的变量具有类的特征，引入包装类。

2.基本数据类型与对应的包装类：

基本数据类型	包装类
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean
char	Character

父类:Number

3.需要掌握的类型间的转换：（基本数据类型、包装类、String）



简易版：

基本数据类型<--->包装类： **JDK 5.0 新特性：自动装箱 与 自动拆箱**

基本数据类型、包装类--->String: 调用String重载的 `valueOf(Xxx Xxx)`

String--->基本数据类型、包装类: 调用包装类的 `parseXxx(String s)`

注意：转换时，可能会报 `NumberFormatException`

应用场景举例：

① Vector类中关于添加元素，只定义了形参为Object类型的方法：

关键字: static

```
v.addElement(Object obj); //基本数据类型 --->包装类 --->使用多态
```

关键字: static

static:静态的

1. 可以用来修饰的结构: 主要用来修饰类的内部结构

属性、方法、代码块、内部类

2. static修饰属性: 静态变量 (或类变量)

2.1 属性, 是否使用static修饰, 又分为: 静态属性 vs 非静态属性(实例变量)

* 实例变量: 我们创建了类的多个对象, 每个对象都独立的拥一套类中的非静态属性。当修改其中一个对象中的非静态属性时, 不会导致其他对象中同样的属性值的修改。

* 静态变量: 我们创建了类的多个对象, 多个对象共享同一个静态变量。当通过某一个对象修改静态变量时, 会导致其他对象调用此静态变量时, 是修改过了的。

2.2 static修饰属性的其他说明:

① 静态变量随着类的加载而加载。可以通过"类.静态变量"的方式进行调用

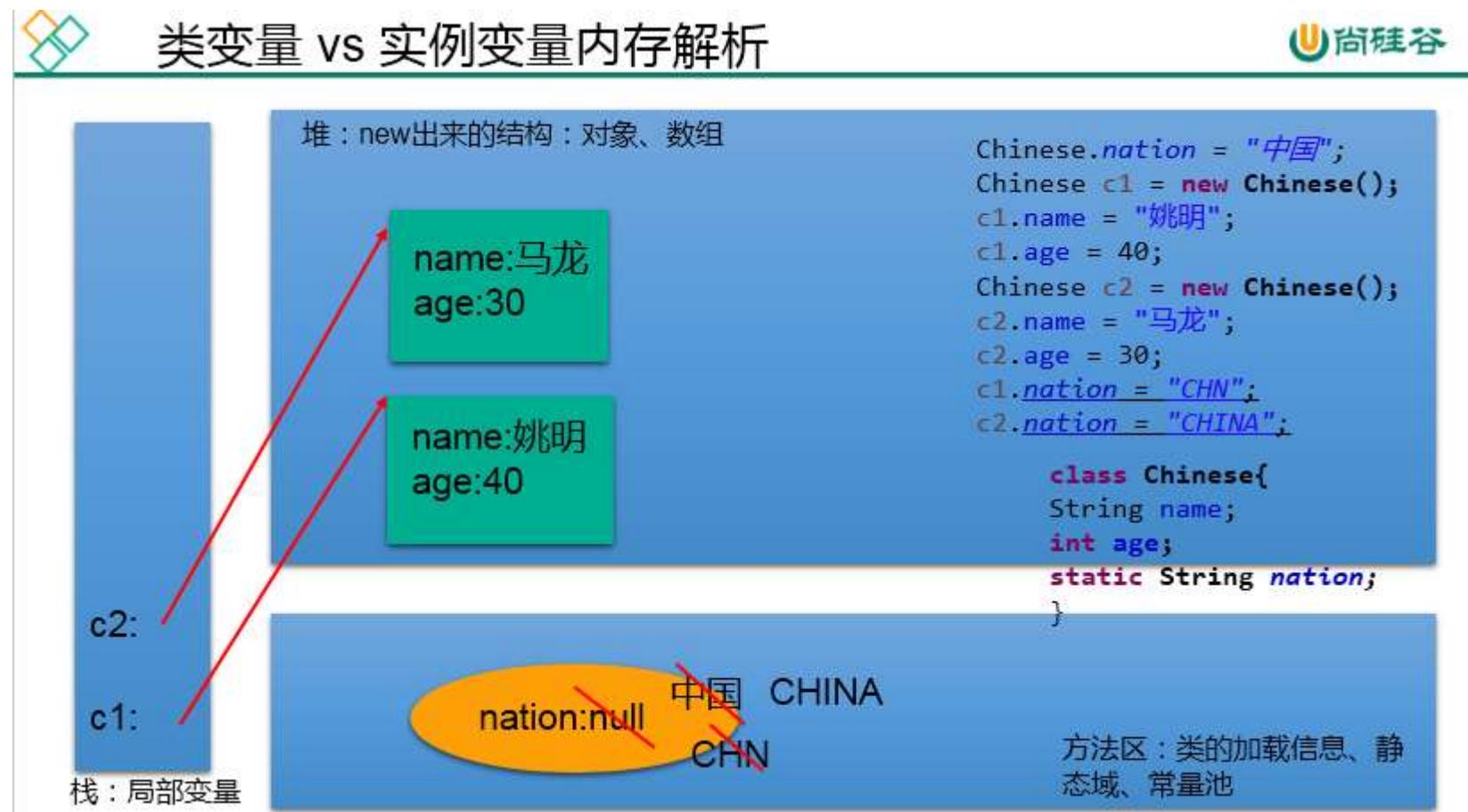
② 静态变量的加载要早于对象的创建。

③ 由于类只会加载一次, 则静态变量在内存中也只会存在一份: 存在方法区的静态域中。

④	类变量	实例变量
类	yes	no
对象	yes	yes

2.3 静态属性举例: System.out; Math.PI;

3. 静态变量内存解析:



4. static修饰方法: 静态方法、类方法

① 随着类的加载而加载, 可以通过"类.静态方法"的方式进行调用

② 静态方法 非静态方法

* 类 yes no

* 对象 yes yes

③ 静态方法中, 只能调用静态的方法或属性

非静态方法中, 既可以调用非静态的方法或属性, 也可以调用静态的方法或属性

5. static的注意点:

5.1 在静态的方法内, 不能使用this关键字、super关键字

5.2 关于静态属性和静态方法的使用, 大家都从生命周期的角度去理解。

6. 如何判定属性和方法应该使用static关键字:

6.1 关于属性

- > 属性是可以被多个对象所共享的，不会随着对象的不同而不同的。
- > 类中的常量也常常声明为static

6.2 关于方法

- > 操作静态属性的方法，通常设置为static的
- > 工具类中的方法，习惯上声明为static的。比如：Math、Arrays、Collections

7. 使用举例：

举例一： Arrays、Math、Collections等工具类

举例二： 单例模式

举例三：

```
class Circle{  
  
    private double radius;  
    private int id;//自动赋值  
  
    public Circle(){  
        id = init++;  
        total++;  
    }  
  
    public Circle(double radius){  
        this();  
        id = init++;  
        total++;  
        this.radius = radius;  
    }  
  
    private static int total;//记录创建的圆的个数  
    private static int init = 1001;//static声明的属性被所有对象所共享  
  
    public double findArea(){  
        return 3.14 * radius * radius;  
    }  
  
    public double getRadius() {  
        return radius;  
    }  
  
    public void setRadius(double radius) {  
        this.radius = radius;  
    }  
  
    public int getId() {  
        return id;  
    }  
  
    public static int getTotal() {  
        return total;  
    }  
}
```

1. 设计模式的说明

1.1 理解

设计模式是在大量的实践中总结和理论化之后优的代码结构、编程风格、以及解决问题的思考方式。

1.2 常用设计模式 --- 23种经典的设计模式 GOF

创建型模式，共5种：工厂方法模式、抽象工厂模式、单例模式、建造者模式、原型模式。

结构型模式，共7种：适配器模式、装饰器模式、代理模式、外观模式、桥接模式、组合模式、享元模式。

行为型模式，共11种：策略模式、模板方法模式、观察者模式、迭代器模式、责任链模式、命令模式、备忘录模式、状态模式、访问者模式、中介者模式、解释器模式。

2. 单例模式

2.1 要解决的问题：

所谓类的单例设计模式，就是采取一定的方法保证在整个的软件系统中，对某个类只能存在一个对象实例。

2.2 具体代码的实现：

饿汉式1：

```
class Bank{
    //1.私化类的构造器
    private Bank(){
    }

    //2.内部创建类的对象
    //4.要求此对象也必须声明为静态的
    private static Bank instance = new Bank();

    //3.提供公共的静态的方法，返回类的对象
    public static Bank getInstance(){
        return instance;
    }
}
```

饿汉式2：使用了静态代码块

```
class Order{
    //1.私化类的构造器
    private Order(){
    }

    //2.声明当前类对象，没初始化
    //4.此对象也必须声明为static的
    private static Order instance = null;

    static{
        instance = new Order();
    }

    //3.声明public、static的返回当前类对象的方法
    public static Order getInstance(){
        return instance;
    }
}
```

懒汉式：

```
class Order{  
    //1.私化类的构造器  
    private Order(){  
    }  
  
    //2.声明当前类对象，没初始化  
    //4.此对象也必须声明为static的  
    private static Order instance = null;  
  
    //3.声明public、static的返回当前类对象的方法  
    public static Order getInstance(){  
  
        if(instance == null){  
            instance = new Order();  
        }  
        return instance;  
    }  
}
```

2.3 两种方式的对比：

- * 饿汉式：
 - * 坏处：对象加载时间过长。
 - * 好处：饿汉式是线程安全的
 - *
- * 懒汉式：好处：延迟对象的创建。
 - * 目前的写法坏处：线程不安全。--->到多线程内容时，再修改

main()的使用说明

- * 1. main()方法作为程序的入口
- * 2. main()方法也是一个普通的静态方法
- * 3. main()方法可以作为我们与控制台交互的方式。 (之前: 使用Scanner)

如何将控制台获取的数据传给形参: String[] args?

运行时: java 类名 "Tom" "Jerry" "123" "true"

```
sysout(args[0]); // "Tom"
sysout(args[3]); // "true" --> Boolean.parseBoolean(args[3]);
sysout(args[4]); // 报异常
```

小结: 一叶知秋

```
public static void main(String[] args){ //方法体}
```

权限修饰符: private 缺省 protected public ----> 封装性

修饰符: static \ final \ abstract \ native 可以用来修饰方法

返回值类型: 无返回值 / 有返回值 --> return

方法名: 需要满足标识符命名的规则、规范; "见名知意"

形参列表: 重载 vs 重写; 参数的值传递机制; 体现对象的多态性

方法体: 来体现方法的功能

```
main() {
    Person p = new Man();
    p.eat();
    // p.earnMoney();

    Man man = new Man();
    man.eat();
    man.earnMoney();
}
```

类的成员之四：代码块(初始化块)（重要性较属性、方法、构造器差一些）

1. 代码块的作用：用来初始化类、对象的信息

2. 分类：代码块要是使用修饰符，只能使用static

分类：静态代码块 vs 非静态代码块

3.

静态代码块：

- >内部可以输出语句
- >随着类的加载而执行，而且只执行一次
- >作用：初始化类的信息
- >如果一个类中定义了多个静态代码块，则按照声明的先后顺序执行
- >静态代码块的执行要优先于非静态代码块的执行
- >静态代码块内只能调用静态的属性、静态的方法，不能调用非静态的结构

非静态代码块：

- >内部可以输出语句
- >随着对象的创建而执行
- >每创建一个对象，就执行一次非静态代码块
- >作用：可以在创建对象时，对对象的属性等进行初始化
- >如果一个类中定义了多个非静态代码块，则按照声明的先后顺序执行
- >非静态代码块内可以调用静态的属性、静态的方法，或非静态的属性、非静态的方法

4. 实例化子类对象时，涉及到父类、子类中静态代码块、非静态代码块、构造器的加载顺序：

对应的练习：`LeafTest.java / Son.java`

由父及子，静态先行。

属性的赋值顺序

- * ①默认初始化
- * ②显式初始化/⑤在代码块中赋值
- * ③构造器中初始化
- * ④有了对象以后，可以通过"对象.属性"或"对象.方法"的方式，进行赋值
- *
- *
- * 执行的先后顺序： ① - ② / ⑤ - ③ - ④

关键字: final

final: 最终的

1. 可以用来修饰: 类、方法、变量

2. 具体的:

2.1 **final** 用来修饰一个类: 此类不能被其他类所继承。

* 比如: `String`类、`System`类、`StringBuffer`类
*

2.2 **final** 用来修饰方法: 表明此方法不可以被重写

* 比如: `Object`类中`getClass()`;
*

2.3 **final** 用来修饰变量: 此时的"变量"就称为是一个常量

* 1. **final**修饰属性: 可以考虑赋值的位置: 显式初始化、代码块中初始化、构造器中初始化

* 2. **final**修饰局部变量:

* 尤其是使用**final**修饰形参时, 表明此形参是一个常量。当我们调用此方法时, 给常量形参赋一个实参。一旦赋值以后, 就只能在方法体内使用此形参, 但不能进行重新赋值。

*

static final 用来修饰属性: 全局常量

关键字: abstract

abstract: 抽象的

1. 可以用来修饰: 类、方法

2. 具体的:

abstract修饰类: 抽象类
 * > 此类不能实例化
 * > 抽象类中一定有构造器, 便于子类实例化时调用 (涉及: 子类对象实例化的全过程)
 * > 开发中, 都会提供抽象类的子类, 让子类对象实例化, 完成相关的操作 ---> 抽象的使用前提: 继承性

abstract修饰方法: 抽象方法
 * > 抽象方法只方法的声明, 没方法体
 * > 包含抽象方法的类, 一定是一个抽象类。反之, 抽象类中可以没有抽象方法的。
 * > 若子类重写了父类中的所有的抽象方法后, 此子类方可实例化
 * 若子类没重写父类中的所有的抽象方法, 则此子类也是一个抽象类, 需要使用**abstract**修饰

3. 注意点:

- * 1. **abstract**不能用来修饰: 属性、构造器等结构
- * 2. **abstract**不能用来修饰私方法、静态方法、**final**的方法、**final**的类

4. **abstract**的应用举例:

举例一:

```
public abstract class Vehicle{
    public abstract double calcFuelEfficiency(); //计算燃料效率的抽象方法
    public abstract double calcTripDistance(); //计算行驶距离的抽象方法
}
public class Truck extends Vehicle{
    public double calcFuelEfficiency() { //写出计算卡车的燃料效率的具体方法 }
    public double calcTripDistance() { //写出计算卡车行驶距离的具体方法 }
}
public class RiverBarge extends Vehicle{
    public double calcFuelEfficiency() { //写出计算驳船的燃料效率的具体方法 }
    public double calcTripDistance() { //写出计算驳船行驶距离的具体方法 }
}
```

举例二:

```
abstract class GeometricObject{
    public abstract double findArea();
}

class Circle extends GeometricObject{
    private double radius;
    public double findArea(){
        return 3.14 * radius * radius;
    }
}
```

举例三: IO流中设计到的抽象类: **InputStream/OutputStream / Reader /Writer**。

在其内部

定义了抽象的**read()**、**write()**方法。

1. 解决的问题

在软件开发中实现一个算法时，整体步骤很固定、通用，这些步骤已经在父类中写好了。但是某些部分易变，易变部分可以抽象出来，供不同子类实现。这就是一种模板模式。

2. 举例

```
abstract class Template{
    //计算某段代码执行所需要花费的时间
    public void spendTime(){
        long start = System.currentTimeMillis();
        this.code(); //不确定的部分、易变的部分
        long end = System.currentTimeMillis();
        System.out.println("花费的时间为：" + (end - start));
    }

    public abstract void code();
}

class SubTemplate extends Template{
    @Override
    public void code() {
        for(int i = 2; i <= 1000; i++){
            boolean isFlag = true;
            for(int j = 2; j <= Math.sqrt(i); j++){
                if(i % j == 0){
                    isFlag = false;
                    break;
                }
            }
            if(isFlag){
                System.out.println(i);
            }
        }
    }
}
```

3. 应用场景

模板方法设计模式是编程中经常用得到的模式。各个框架、类库中都有他的影子，比如常见的有：

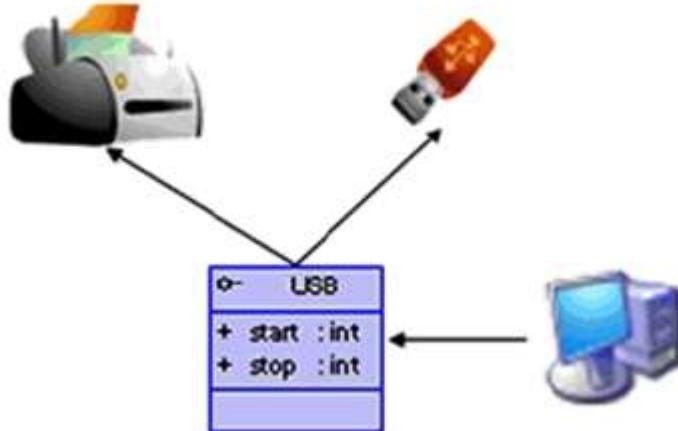
- 数据库访问的封装
- Junit单元测试
- JavaWeb的Servlet中关于doGet/doPost方法调用
- Hibernate中模板程序
- Spring中JDBCTemplate、HibernateTemplate等

interface: 接口

1. 使用说明:

- 1. 接口使用 **interface** 来定义
- * 2. Java 中，接口和类是并列的两个结构
- * 3. 如何定义接口：定义接口中的成员
- *
- * 3.1 JDK7 及以前：只能定义全局常量和抽象方法
 - > 全局常量：public static final 的。但是书写时，可以省略不写
 - > 抽象方法：public abstract 的
- *
- * 3.2 JDK8：除了定义全局常量和抽象方法之外，还可以定义静态方法、默认方法（略）
- *
- * 4. 接口中不能定义构造器的！意味着接口不可以实例化
- *
- * 5. Java 开发中，接口通过让类去实现 (**implements**) 的方式来使用。
 - 如果实现类覆盖了接口中的所有抽象方法，则此实现类就可以实例化
 - 如果实现类没覆盖接口中所有的抽象方法，则此实现类仍为一个抽象类
- *
- * 6. Java 类可以实现多个接口 ----> 弥补了 Java 单继承性的局限性
 - 格式：class AA extends BB implements CC, DD, EE
- *
- * 7. 接口与接口之间可以继承，而且可以多继承
- *
- * ****
- * 8. 接口的具体使用，体现多态性
- * 9. 接口，实际上可以看做是一种规范

2. 举例：



```

class Computer{

    public void transferData(USB usb){ //USB usb = new Flash();
        usb.start();

        System.out.println("具体传输数据的细节");

        usb.stop();
    }

}

interface USB{
    //常量：定义了长、宽、最大最小的传输速度等

    void start();

    void stop();

}

class Flash implements USB{

    @Override
    public void start() {
        System.out.println("U盘开启工作");
    }

}
  
```

```

@Override
public void stop() {
    System.out.println("U盘结束工作");
}

}

class Printer implements USB{
@Override
public void start() {
    System.out.println("打印机开启工作");
}

@Override
public void stop() {
    System.out.println("打印机结束工作");
}

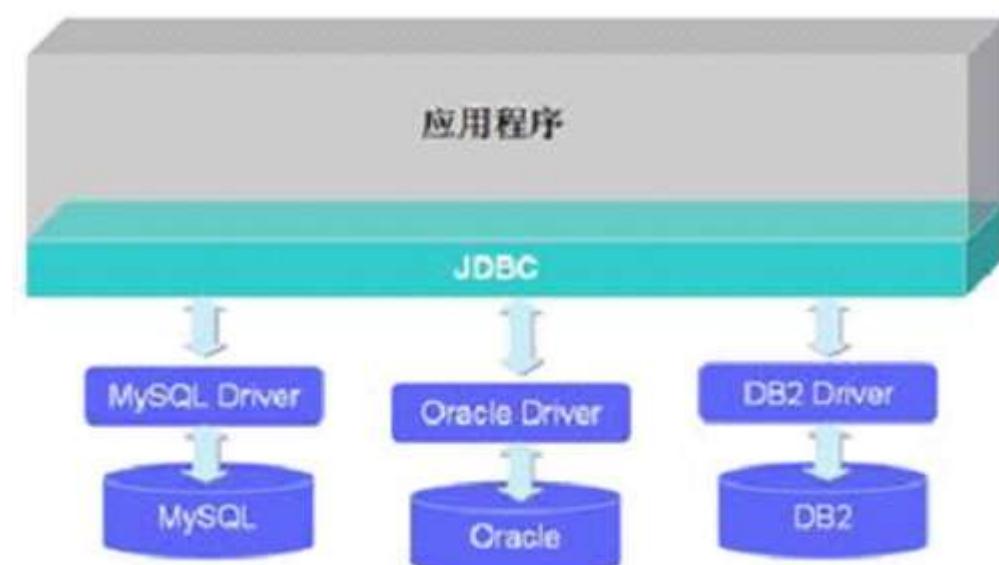
}

```

体会：

- * 1. 接口使用上也满足多态性
- * 2. 接口，实际上就是定义了一种规范
- * 3. 开发中，体会面向接口编程！

3. 体会面向接口编程的思想



面向接口编程：我们在应用程序中，调用的结构都是JDBC中定义的接口，不会出现具体某一个

数据库厂商的API。

4. Java8中关于接口的新规范

//知识点1：接口中定义的静态方法，只能通过接口来调用。

//知识点2：通过实现类的对象，可以调用接口中的默认方法。

//如果实现类重写了接口中的默认方法，调用时，仍然调用的是重写以后的方法

//知识点3：如果子类(或实现类)继承的父类和实现的接口中声明了同名同参数的默认方法，那么子类在没重写此方法的情况下，默认调用的是父类中的同名同参数的方法。-->类优先原则

//知识点4：如果实现类实现了多个接口，而这多个接口中定义了同名同参数的默认方法，那么在实现类没重写此方法的情况下，报错。-->接口冲突。

//这就需要我们必须在实现类中重写此方法

//知识点5：如何在子类(或实现类)的方法中调用父类、接口中被重写的方法

```

public void myMethod(){
    method3(); //调用自己定义的重写的方法
    super.method3(); //调用的是父类中声明的
    //调用接口中的默认方法
    CompareA.super.method3();
    CompareB.super.method3();
}

```

5. 面试题：

抽象类和接口的异同?

相同点: 不能实例化; 都可以包含抽象方法的。

不同点:

1) 把抽象类和接口(java7,java8,java9)的定义、内部结构解释说明

2) 类: 单继承性 接口: 多继承

类与接口: 多实现

1. 解决的问题

代理模式是Java开发中使用较多的一种设计模式。代理设计就是为其他对象提供一种代理以控制对这个对象的访问。

2. 举例

```
interface Network{  
    public void browse();  
}  
  
//被代理类  
class Server implements Network{  
  
    @Override  
    public void browse() {  
        System.out.println("真实的服务器访问网络");  
    }  
  
}  
//代理类  
class ProxyServer implements Network{  
  
    private Network work;  
  
    public ProxyServer(Network work){  
        this.work = work;  
    }  
  
    public void check(){  
        System.out.println("联网之前的检查工作");  
    }  
  
    @Override  
    public void browse() {  
        check();  
  
        work.browse();  
    }  
}
```

3. 应用场景

● 应用场景：

- 安全代理：屏蔽对真实角色的直接访问。
- 远程代理：通过代理类处理远程方法调用（RMI）
- 延迟加载：先加载轻量级的代理对象，真正需要再加载真实对象

比如你要开发一个大文档查看软件，大文档中有大的图片，有可能一个图片有100MB，在打开文件时，不可能将所有的图片都显示出来，这样就可以使用代理模式，当需要查看图片时，用proxy来进行大图片的打开。

● 分类

- 静态代理（静态定义代理类）
- 动态代理（动态生成代理类）
 - ✓ JDK自带的动态代理，需要反射等知识

工厂的设计模式

1. 解决的问题

实现了创建者与调用者的分离，即将创建对象的具体过程屏蔽隔离起来，达到提高灵活性的目的。

2. 具体模式

- **简单工厂模式：**用来生产同一等级结构中的任意产品。（对于增加新的产品，需要修改已有代码）
- **工厂方法模式：**用来生产同一等级结构中的固定产品。（支持增加任意产品）
- **抽象工厂模式：**用来生产不同产品族的全部产品。（对于增加新的产品，无能为力；支持增加产品族）

内部类：类的第五个成员

1. 定义：Java中允许将一个类A声明在另一个类B中，则类A就是内部类，类B称为外部类。

2. 内部类的分类：

成员内部类（静态、非静态） vs 局部内部类(方法内、代码块内、构造器内)

3. 成员内部类的理解：

一方面，作为外部类的成员：

- * > 调用外部类的结构
- * > 可以被static修饰
- * > 可以被4种不同的权限修饰
- *

另一方面，作为一个类：

- * > 类内可以定义属性、方法、构造器等
- * > 可以被final修饰，表示此类不能被继承。言外之意，不使用final，就可以被继承
- * > 可以被abstract修饰

4. 成员内部类：

4.1 如何创建成员内部类的对象？（静态的，非静态的）

//创建静态的Dog内部类的实例(静态的成员内部类)：

```
Person.Dog dog = new Person.Dog();
```

//创建非静态的Bird内部类的实例(非静态的成员内部类)：

```
//Person.Bird bird = new Person.Bird(); //错误的
```

```
Person p = new Person();
```

```
Person.Bird bird = p.new Bird();
```

4.2 如何在成员内部类中调用外部类的结构？

```
class Person{
    String name = "小明";
    public void eat(){
    }
    //非静态成员内部类
    class Bird{
        String name = "杜鹃";
        public void display(String name){
            System.out.println(name); //方法的形参
            System.out.println(this.name); //内部类的属性
            System.out.println(Person.this.name); //外部类的属性
            //Person.this.eat();
        }
    }
}
```

5. 局部内部类的使用：

//返回一个实现了Comparable接口的类的对象

```
public Comparable getComparable(){
```

//创建一个实现了Comparable接口的类：局部内部类

//方式一：

```
//      class MyComparable implements Comparable{
//
```

```
//          @Override
//          public int compareTo(Object o) {
//              return 0;
//          }
//
//      }
//
//      return new MyComparable();

//方式二：
return new Comparable(){

    @Override
    public int compareTo(Object o) {
        return 0;
    }

};
```

注意点：

在局部内部类的方法中（比如：`show`如果调用局部内部类所声明的方法（比如：`method`）中的局部变量（比如：`num`）的话，要求此局部变量声明为`final`的。

* `jdk 7及之前版本：要求此局部变量显式的声明为final的`

`jdk 8及之后的版本：可以省略final的声明`

总结：

成员内部类和局部内部类，在编译以后，都会生成字节码文件。

格式：成员内部类：外部类\$内部类名.class

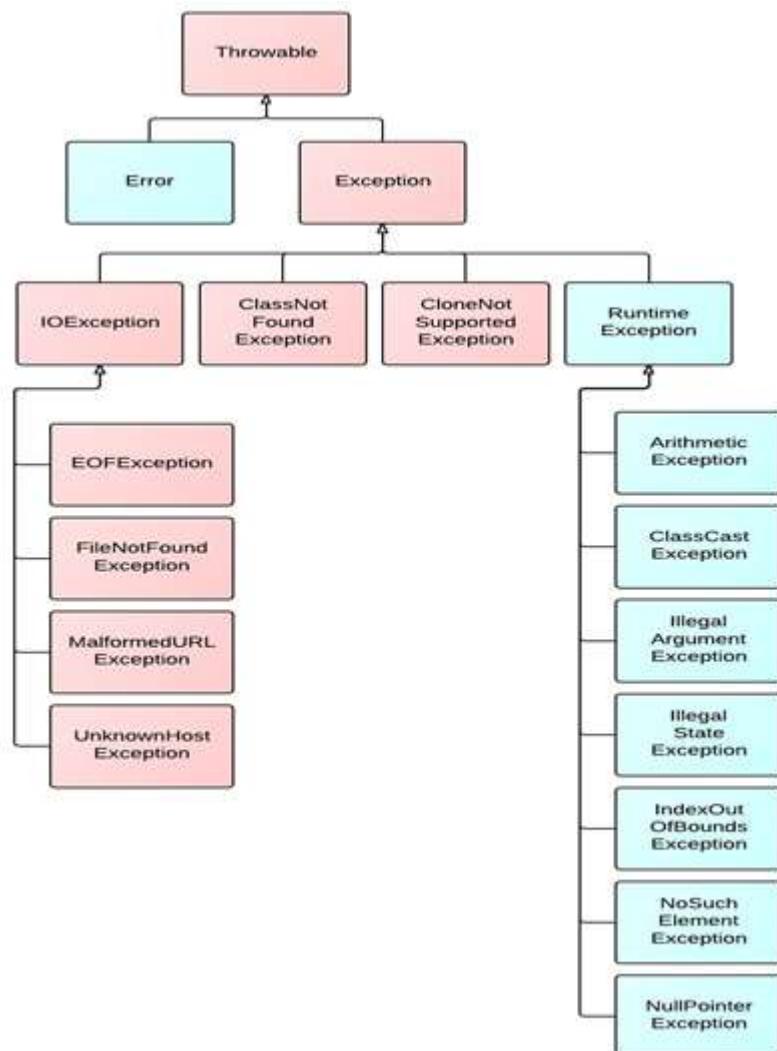
局部内部类：外部类\$数字 内部类名.class

1. 异常的体系结构

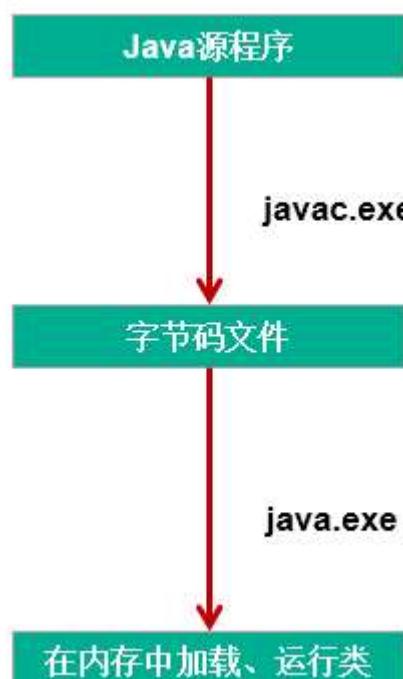
```

* java.lang.Throwable
*   |----java.lang.Error:一般不编写针对性的代码进行处理。
*   |----java.lang.Exception:可以进行异常的处理
*     |----编译时异常(checked)
*       |----IOException
*         |----FileNotFoundException
*         |----ClassNotFoundException
*     |----运行时异常(unchecked, RuntimeException)
*       |----NullPointerException
*       |----ArrayIndexOutOfBoundsException
*       |----ClassCastException
*       |----NumberFormatException
*       |----InputMismatchException
*       |----ArithmaticException

```



2. 从程序执行过程，看编译时异常和运行时异常



编译时异常：执行javac.exe命名时，可能出现的异常

运行时异常：执行java.exe命名时，出现的异常

3. 常见的异常类型，请举例说明：

```
//*****以下就是运行时异常*****
//ArithmException
@Test
public void test6(){
    int a = 10;
    int b = 0;
    System.out.println(a / b);
}

//InputMismatchException
@Test
public void test5(){
    Scanner scanner = new Scanner(System.in);
    int score = scanner.nextInt();
    System.out.println(score);

    scanner.close();
}

//NumberFormatException
@Test
public void test4(){

    String str = "123";
    str = "abc";
    int num = Integer.parseInt(str);

}

//ClassCastException
@Test
public void test3(){
    Object obj = new Date();
    String str = (String) obj;
}

//IndexOutOfBoundsException
@Test
public void test2(){
    //ArrayIndexOutOfBoundsException
    // int[] arr = new int[10];
    // System.out.println(arr[10]);
    //StringIndexOutOfBoundsException
    String str = "abc";
    System.out.println(str.charAt(3));
}

//NullPointerException
@Test
```

```
public void test1(){

    //     int[] arr = null;
    //     System.out.println(arr[3]);

    String str = "abc";
    str = null;
    System.out.println(str.charAt(0));

}

//*****以下就是编译时异常*****
@Test
public void test7(){
    File file = new File("hello.txt");
    FileInputStream fis = new FileInputStream(file);
    int data = fis.read();
    while(data != -1){
        System.out.print((char)data);
        data = fis.read();
    }
    fis.close();

}
```

1.java异常处理的抓抛模型

过程一：“抛”：程序在正常执行的过程中，一旦出现异常，就会在异常代码处生成一个对应异常类的对象。

- * 并将此对象抛出。
- * 一旦抛出对象以后，其后的代码就不再执行。

- * 关于异常对象的产生：^① 系统自动生成的异常对象
- * ^② 手动的生成一个异常对象，并抛出（`throw`）

过程二：“抓”：可以理解为异常的处理方式：^① `try-catch-finally` ^② `throws`

2.异常处理方式一：try-catch-finally

2.1 使用说明：

```
try{
    *      //可能出现异常的代码
    *
    * }catch(异常类型1 变量名1){
    *     //处理异常的方式1
    * }catch(异常类型2 变量名2){
    *     //处理异常的方式2
    * }catch(异常类型3 变量名3){
    *     //处理异常的方式3
    * }
    * ....
    * finally{
    *     //一定会执行的代码
    * }
    *
    * 说明：
    * 1. finally是可的。
    * 2. 使用try将可能出现异常代码包装起来，在执行过程中，一旦出现异常，就会生成一个对应异常类的对象，根据此对象的类型，去catch中进行匹配
    * 3. 一旦try中的异常对象匹配到某一个catch时，就进入catch中进行异常的处理。一旦处理完成，就跳出当前的try-catch结构（在没写finally的情况下）。继续执行其后的代码
    * 4. catch中的异常类型如果没子父类关系，则谁声明在上，谁声明在下无所谓。
    *   catch中的异常类型如果满足子父类关系，则要求子类一定声明在父类的上面。否则，报错
    * 5. 常用的异常对象处理的方式：① String getMessage() ② printStackTrace()
    * 6. 在try结构中声明的变量，再出了try结构以后，就不能再被调用
    * 7. try-catch-finally结构可以嵌套
```

总结：如何看待代码中的编译时异常和运行时异常？

- * 体会1：使用try-catch-finally处理编译时异常，使得程序在编译时就不再报错，但是运行时仍可能报错。相当于我们使用try-catch-finally将一个编译时可能出现的异常，延迟到运行时出现。

- * 体会2：开发中，由于运行时异常比较常见，所以我们通常就不针对运行时异常编写try-catch-finally了。而对于编译时异常，我们说一定要考虑异常的处理。

2.2: finally的再说明：

- * 1. finally是可的
- *
- * 2. finally中声明的是一定会被执行的代码。即使catch中又出现异常了，try中return语句，catch中return语句等情况。
- *

* 3. 像数据库连接、输入输出流、网络编程Socket等资源，JVM是不能自动的回收的，我们需要自己手动的进行资源的释放。此时的资源释放，就需要声明在**finally**中。

2.3: [面试题]

final、**finally**、**finalize**三者的区别？

类似：

throw 和 **throws**

Collection 和 **Collections**

String 、**StringBuffer**、**StringBuilder**

ArrayList 、**LinkedList**

HashMap 、**LinkedHashMap**

重写、重载

结构不相似的：

抽象类、接口

== 、 **equals()**

sleep()、**wait()**

3. 异常处理方式二：

"**throws** + 异常类型"写在方法的声明处。指明此方法执行时，可能会抛出的异常类型。

一旦当方法体执行时，出现异常，仍会在异常代码处生成一个异常类的对象，此对象满足**throws**后异常类型时，就会被抛出。异常代码后续的代码，就不再执行！

4. 对比两种处理方式

try-catch-finally: 真正的将异常给处理掉了。

throws的方式只是将异常抛给了方法的调用者。并没真正将异常处理掉。

5. 体会开发中应该如何选择两种处理方式？

* 5.1 如果父类中被重写的方法没**throws**方式处理异常，则子类重写的方法也不能使用**throws**，意味着如果子类重写的方法中异常，必须使用**try-catch-finally**方式处理。

* 5.2 执行的方法a中，先后又调用了另外的几个方法，这几个方法是递进关系执行的。我们建议这几个方法使用**throws**的方式进行处理。而执行的方法a可以考虑使用**try-catch-finally**方式进行处理。

补充：

方法重写的规则之一：

子类重写的方法抛出的异常类型不大于父类被重写的方法抛出的异常类型

手动抛出异常对象

1. 使用说明

在程序执行中，除了自动抛出异常对象的情况之外，我们还可以手动的**throw**一个异常类的对象。

2. [面试题]

throw 和 **throws**区别：

throw 表示抛出一个异常类的对象，生成异常对象的过程。声明在方法体内。

throws 属于异常处理的一种方式，声明在方法的声明处。

3. 典型例题

```
class Student{  
  
    private int id;  
  
    public void regist(int id) throws Exception {  
        if(id > 0){  
            this.id = id;  
        }else{  
            //手动抛出异常对象  
            throw new RuntimeException("您输入的数据非法！");  
            throw new Exception("您输入的数据非法！");  
            throw new MyException("不能输入负数");  
        }  
    }  
  
    @Override  
    public String toString() {  
        return "Student [id=" + id + "]";  
    }  
  
}
```

自定义异常类

如何自定义一个异常类？

```
/*
 * 如何自定义异常类?
 * 1. 继承于现的异常结构: RuntimeException 、 Exception
 * 2. 提供全局常量: serialVersionUID
 * 3. 提供重载的构造器
 *
 */
public class MyException extends Exception{

    static final long serialVersionUID = -7034897193246939L;

    public MyException(){

    }

    public MyException(String msg){
        super(msg);
    }
}
```

01. 程序(programm)

概念：是为完成特定任务、用某种语言编写的一组指令的集合。即指一段静态的代码。

02. 进程(process)

概念：程序的一次执行过程，或是正在运行的一个程序。

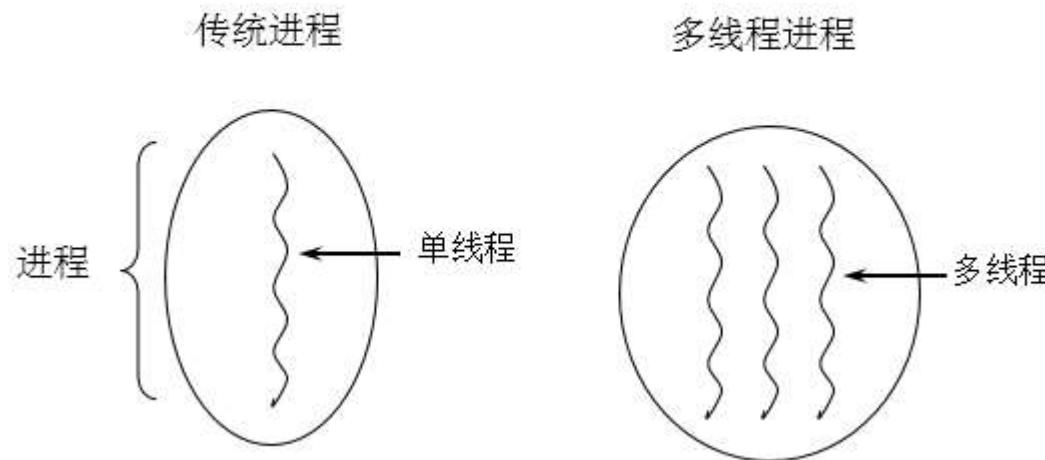
说明：进程作为资源分配的单位，系统在运行时会为每个进程分配不同的内存区域

03. 线程(thread)

概念：进程可进一步细化为线程，是一个程序内部的一条执行路径。

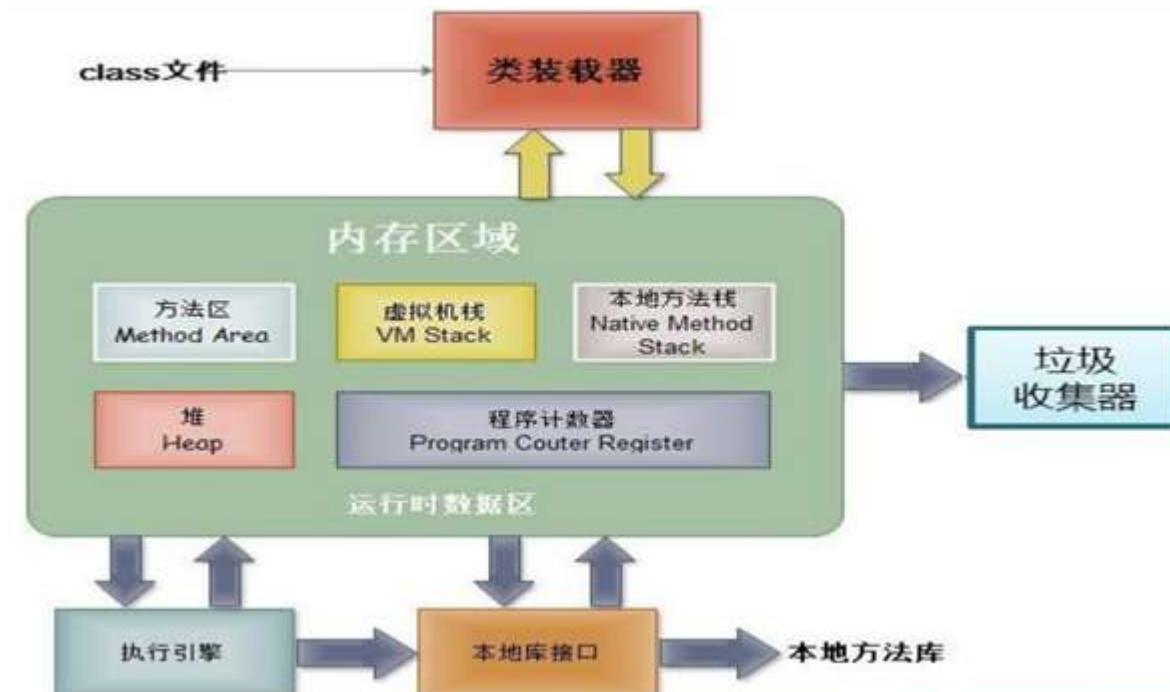
说明：线程作为调度和执行的单位，每个线程拥独立的运行栈和程序计数器(pc)，线程切换的开销小。

进程与线程



补充：

内存结构：



进程可以细化为多个线程。

每个线程，拥有自己独立的：栈、程序计数器

多个线程，共享同一个进程中的结构：方法区、堆。

01. 单核CPU与多核CPU的理解

- ① **单核CPU**, 其实是一种假的多线程, 因为在一个时间单元内, 也只能执行一个线程的任务。例如: 虽然有多车道, 但是收费站只有一个工作人员在收费, 只有收了费才能通过, 那么**CPU**就好比收费人员。如果某个人不想交钱, 那么收费人员可以把他“挂起”(晾着他, 等他想通了, 准备好了钱, 再去收费。)但是因为**CPU**时间单元特别短, 因此感觉不出来。
- ② 如果是多核的话, 才能更好的发挥多线程的效率。(现在的服务器都是多核的)
- ③ 一个Java应用程序**java.exe**, 其实至少三个线程: `main()`主线程, `gc()`垃圾回收线程, 异常处理线程。当然如果发生异常, 会影响主线程。

02. 并行与并发的理解

- ① **并行**: 多个CPU同时执行多个任务。比如: 多个人同时做不同的事。
- ② **并发**: 一个CPU(采用时间片)同时执行多个任务。比如: 秒杀、多个人做同一件事

创建多线程的两种方式

方式一：继承Thread类的方式：

- * 1. 创建一个继承于Thread类的子类
- * 2. 重写Thread类的run() --> 将此线程执行的操作声明在run()中
- * 3. 创建Thread类的子类的对象
- * 4. 通过此对象调用start(): ①启动当前线程 ② 调用当前线程的run()

说明两个问题：

问题一：我们启动一个线程，必须调用start()，不能调用run()的方式启动线程。

问题二：如果再启动一个线程，必须重新创建一个Thread子类的对象，调用此对象的start()。

方式二：实现Runnable接口的方式：

- * 1. 创建一个实现了Runnable接口的类
- * 2. 实现类去实现Runnable中的抽象方法：run()
- * 3. 创建实现类的对象
- * 4. 将此对象作为参数传递到Thread类的构造器中，创建Thread类的对象
- * 5. 通过Thread类的对象调用start()

两种方式的对比：

- * 开发中：优先选择：实现Runnable接口的方式
- * 原因：
 - 1. 实现的方式没类的单继承性的局限性
 - 2. 实现的方式更适合来处理多个线程共享数据的情况。
- *
- * 联系：public class Thread implements Runnable
- * 相同点：两种方式都需要重写run()，将线程要执行的逻辑声明在run()中。
目前两种方式，要想启动线程，都是调用的Thread类中的start()。

Thread类中的常用的方法：

- * 1. `start()`: 启动当前线程；调用当前线程的`run()`
 - * 2. `run()`: 通常需要重写Thread类中的此方法，将创建的线程要执行的操作声明在此方法中
 - * 3. `currentThread()`: 静态方法，返回执行当前代码的线程
 - * 4. `getName()`: 获取当前线程的名字
 - * 5. `setName()`: 设置当前线程的名字
 - * 6. `yield()`: 释放当前cpu的执行权
 - * 7. `join()`: 在线程a中调用线程b的`join()`，此时线程a就进入阻塞状态，直到线程b完全执行完以后，线程a才结束阻塞状态。
 - * 8. `stop()`: 已过时。当执行此方法时，强制结束当前线程。
 - * 9. `sleep(long millitime)`: 让当前线程“睡眠”指定的millitime毫秒。在指定的millitime毫秒时间内，当前线程是阻塞状态。
 - * 10. `isAlive()`: 判断当前线程是否存活
-
- * 线程的优先级：
 - * 1.
 - * `MAX_PRIORITY: 10`
 - * `MIN_PRIORITY: 1`
 - * `NORM_PRIORITY: 5` --> 默认优先级
 - * 2. 如何获取和设置当前线程的优先级：
 - * `getPriority()`: 获取线程的优先级
 - * `setPriority(int p)`: 设置线程的优先级
- * 说明：高优先级的线程要抢占低优先级线程cpu的执行权。但是只是从概率上讲，高优先级的线程高概率的情况下被执行。并不意味着只当高优先级的线程执行完以后，低优先级的线程才执行。

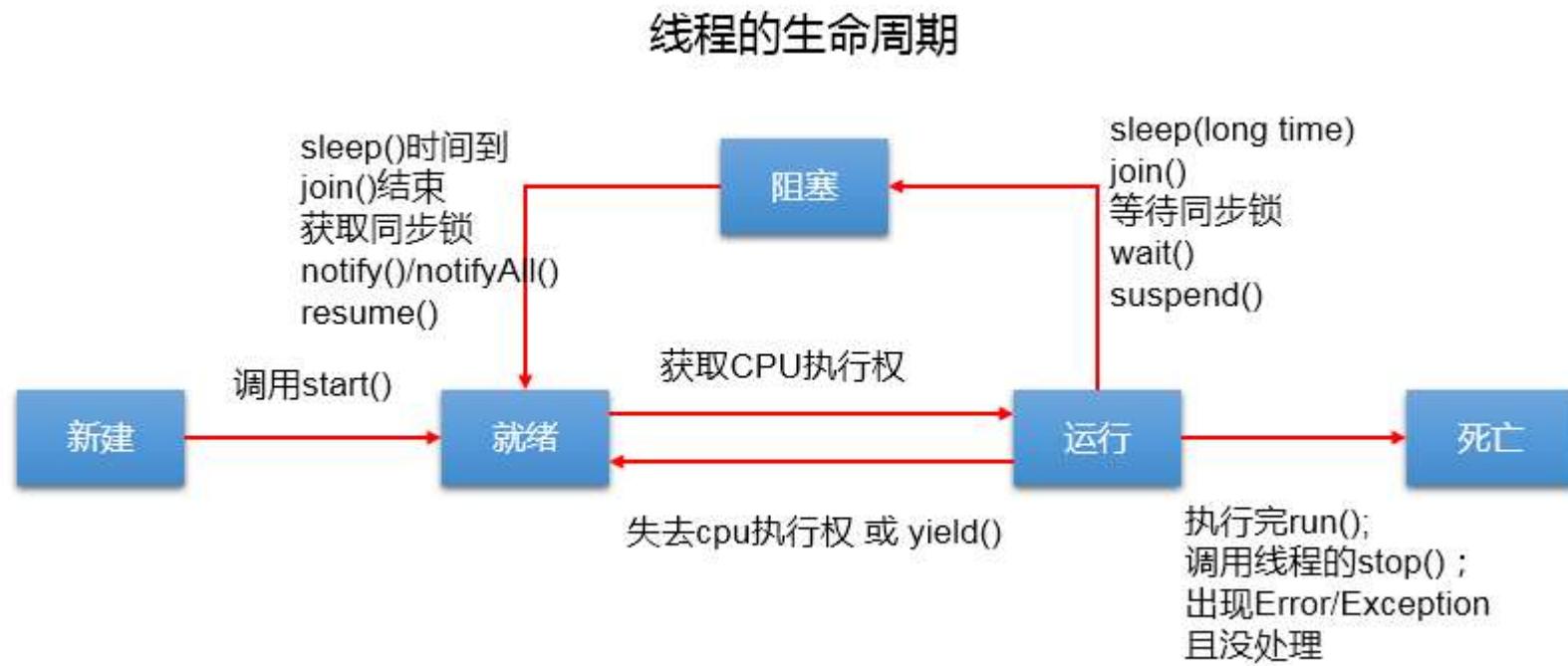
线程通信：`wait()` / `notify()` / `notifyAll()` :此三个方法定义在Object类中的。

补充：线程的分类

一种是**守护线程**，一种是**用户线程**。

Thread的生命周期

图示：



说明：

1. 生命周期关注两个概念：状态、相应的方法
 2. 关注：状态a-->状态b: 哪些方法执行了（回调方法）
某个方法主动调用：状态a-->状态b
 3. 阻塞：临时状态，不可以作为最终状态
- 死亡：最终状态。

1. 背景

例子：创建个窗口卖票，总票数为100张。使用实现`Runnable`接口的方式

- * 1. 问题：卖票过程中，出现了重票、错票 --> 出现了线程的安全问题
- * 2. 问题出现的原因：当某个线程操作车票的过程中，尚未操作完成时，其他线程参与进来，也操作车票。
- * 3. 如何解决：当一个线程a在操作`ticket`的时候，其他线程不能参与进来。直到线程a操作完`ticket`时，其他线程才可以开始操作`ticket`。这种情况即使线程a出现了阻塞，也不能被改变。

2. Java解决方案：同步机制

在Java中，我们通过同步机制，来解决线程的安全问题。

方式一：同步代码块

- *
- * `synchronized(同步监视器){`
- * `//需要被同步的代码`
- *
- * `}`
- * 说明：1. 操作共享数据的代码，即为需要被同步的代码。 --> 不能包含代码多了，也不能包含代码少了。
- * 2. 共享数据：多个线程共同操作的变量。比如：`ticket`就是共享数据。
- * 3. 同步监视器，俗称：锁。任何一个类的对象，都可以充当锁。
- * 要求：多个线程必须要共用同一把锁。
- *
- * 补充：在实现`Runnable`接口创建多线程的方式中，我们可以考虑使用`this`充当同步监视器。

在继承`Thread`类创建多线程的方式中，慎用`this`充当同步监视器，考虑使用当前类充当同步监视器。

方式二：同步方法

- * 如果操作共享数据的代码完整的声明在一个方法中，我们不妨将此方法声明同步的。
- *
- * 关于同步方法的总结：
- * 1. 同步方法仍然涉及到同步监视器，只是不需要我们显式的声明。
- * 2. 非静态的同步方法，同步监视器是：`this`
- * 静态的同步方法，同步监视器是：当前类本身

方式三：Lock 锁 --- JDK5.0新增

- *
- * 1. 面试题：`synchronized` 与 `Lock`的异同？
- * 相同：二者都可以解决线程安全问题
- * 不同：`synchronized`机制在执行完相应的同步代码以后，自动的释放同步监视器
- * `Lock`需要手动的启动同步（`lock()`），同时结束同步也需要手动的实现（`unlock()`）

使用的优先顺序：

* `Lock` ---> 同步代码块（已经进入了方法体，分配了相应资源） ---> 同步方法（在方法体之外）

3. 利弊

同步的方式，解决了线程的安全问题。---好处

操作同步代码时，只能一个线程参与，其他线程等待。相当于是一个单线程的过程，效率低。

4.

面试题：Java是如何解决线程安全问题的，有几种方式？并对比几种方式的不同

面试题： `synchronized`和`Lock`方式解决线程安全问题的对比

线程安全的单例模式(懒汉式)

使用同步机制将单例模式中的懒汉式改写为线程安全的。

```
class Bank{  
  
    private Bank(){}
  
  
    private static Bank instance = null;
  
  
    public static Bank getInstance(){  
        //方式一：效率稍差  
        //      synchronized (Bank.class) {  
        //          if(instance == null){  
        //              instance = new Bank();  
        //          }  
        //          return instance;  
        //      }  
        //方式二：效率更高  
        if(instance == null){  
  
            synchronized (Bank.class) {  
                if(instance == null){  
  
                    instance = new Bank();  
                }  
            }  
        }  
        return instance;  
    }  
}
```

面试题：写一个线程安全的单例模式。

饿汉式。

懒汉式：上面提供的。

1.死锁的理解:

不同的线程分别占用对方需要的同步资源不放弃，都在等待对方放弃自己需要的同步资源，就形成了线程的死锁

2.说明:

* 1出现死锁后，不会出现异常，不会出现提示，只是所有的线程都处于阻塞状态，无法继续

* 2我们使用同步时，要避免出现死锁。

3.举例:

```
public static void main(String[] args) {  
  
    StringBuffer s1 = new StringBuffer();  
    StringBuffer s2 = new StringBuffer();  
  
    new Thread(){  
        @Override  
        public void run() {  
  
            synchronized (s1){  
  
                s1.append("a");  
                s2.append("1");  
  
                try {  
                    Thread.sleep(100);  
                } catch (InterruptedException e) {  
                    e.printStackTrace();  
                }  
  
                synchronized (s2){  
                    s1.append("b");  
                    s2.append("2");  
  
                    System.out.println(s1);  
                    System.out.println(s2);  
                }  
  
            }  
        }.start();  
  
    new Thread(new Runnable() {  
        @Override  
        public void run() {  
            synchronized (s2){  
  
                s1.append("c");  
                s2.append("3");  
  
                try {  
                    Thread.sleep(100);  
                } catch (InterruptedException e) {  
                }  
            }  
        }  
    });  
}
```

```
        e.printStackTrace();
    }

    synchronized (s1){
        s1.append("d");
        s2.append("4");

        System.out.println(s1);
        System.out.println(s2);
    }

}

}).start();

}
```

1. 线程通信涉及到的三个方法：

- * `wait()`: 一旦执行此方法，当前线程就进入阻塞状态，并释放同步监视器。
- * `notify()`: 一旦执行此方法，就会唤醒被`wait`的一个线程。如果有多个线程被`wait`，就唤醒优先级高的那个。
- * `notifyAll()`: 一旦执行此方法，就会唤醒所有被`wait`的线程。

2. 说明：

- * 1. `wait()`, `notify()`, `notifyAll()`三个方法必须使用在同步代码块或同步方法中。
- * 2. `wait()`, `notify()`, `notifyAll()`三个方法的调用者必须是同步代码块或同步方法中的同步监视器。
- * 否则，会出现`IllegalMonitorStateException`异常
- * 3. `wait()`, `notify()`, `notifyAll()`三个方法是定义在`java.lang.Object`类中。

3. 面试题：

面试题：`sleep()` 和 `wait()` 的异同？

- * 1. 相同点：一旦执行方法，都可以使得当前的线程进入阻塞状态。
- * 2. 不同点：1) 两个方法声明的位置不同：`Thread`类中声明`sleep()`，`Object`类中声明`wait()`
- * 2) 调用的要求不同：`sleep()`可以在任何需要的场景下调用。`wait()`必须使用在同步代码块或同步方法中
- * 3) 关于是否释放同步监视器：如果两个方法都使用在同步代码块或同步方法中，`sleep()`不会释放锁，`wait()`会释放锁。

4.

小结释放锁的操作：

释放锁的操作

- 当前线程的同步方法、同步代码块执行结束。
- 当前线程在同步代码块、同步方法中遇到`break`、`return`终止了该代码块、该方法的继续执行。
- 当前线程在同步代码块、同步方法中出现了未处理的`Error`或`Exception`，导致异常结束。
- 当前线程在同步代码块、同步方法中执行了线程对象的`wait()`方法，当前线程暂停，并释放锁。

小结不会释放锁的操作：

不会释放锁的操作

- 线程执行同步代码块或同步方法时，程序调用`Thread.sleep()`、`Thread.yield()`方法暂停当前线程的执行
- 线程执行同步代码块时，其他线程调用了该线程的`suspend()`方法将该线程挂起，该线程不会释放锁（同步监视器）。
 - 应尽量避免使用`suspend()`和`resume()`来控制线程

JDK5.0新增线程创建的方式

新增方式一：实现`Callable`接口。 --- JDK 5.0新增

```
//1. 创建一个实现Callable的实现类
class NumThread implements Callable{
    //2. 实现call方法，将此线程需要执行的操作声明在call()中
    @Override
    public Object call() throws Exception {
        int sum = 0;
        for (int i = 1; i <= 100; i++) {
            if(i % 2 == 0){
                System.out.println(i);
                sum += i;
            }
        }
        return sum;
    }

public class ThreadNew {
    public static void main(String[] args) {
        //3. 创建Callable接口实现类的对象
        NumThread numThread = new NumThread();
        //4. 将此Callable接口实现类的对象作为传递到FutureTask构造器中，创建FutureTask的对象
        FutureTask futureTask = new FutureTask(numThread);
        //5. 将FutureTask的对象作为参数传递到Thread类的构造器中，创建Thread对象，并调用start()
        new Thread(futureTask).start();

        try {
            //6. 获取Callable中call方法的返回值
            //get()返回值即为FutureTask构造器参数Callable实现类重写的call()的返回值。
            Object sum = futureTask.get();
            System.out.println("总和为：" + sum);
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            e.printStackTrace();
        }
    }
}
```

说明：

- * 如何理解实现`Callable`接口的方式创建多线程比实现`Runnable`接口创建多线程方式强大？
- * 1. `call()`可以返回值的。
- * 2. `call()`可以抛出异常，被外面的操作捕获，获取异常的信息
- * 3. `Callable`是支持泛型的

新增方式二：使用线程池

```
class NumberThread implements Runnable{
    @Override
    public void run() {
        for(int i = 0;i <= 100;i++){
            if(i % 2 == 0){
                System.out.println(Thread.currentThread().getName() + ":" + i);
            }
        }
    }
}

class NumberThread1 implements Runnable{
    @Override
    public void run() {
        for(int i = 0;i <= 100;i++){
            if(i % 2 != 0){
```

```

        System.out.println(Thread.currentThread().getName() + ":" + i);
    }
}
}

public class ThreadPool {
    public static void main(String[] args) {
        //1. 提供指定线程数量的线程池
        ExecutorService service = Executors.newFixedThreadPool(10);
        ThreadPoolExecutor service1 = (ThreadPoolExecutor) service;
        //设置线程池的属性
        // System.out.println(service.getClass());
        // service1.setCorePoolSize(15);
        // service1.setKeepAliveTime();

        //2. 执行指定的线程的操作。需要提供实现Runnable接口或Callable接口实现类的对象
        service.execute(new NumberThread());//适合适用于Runnable
        service.execute(new NumberThread1());//适合适用于Runnable

        // service.submit(Callable callable); //适合使用于Callable
        //3. 关闭连接池
        service.shutdown();
    }
}

```

说明：

- * 好处：
- * 1. 提高响应速度（减少了创建新线程的时间）
- * 2. 降低资源消耗（重复利用线程池中线程，不需要每次都创建）
- * 3. 便于线程管理
- * corePoolSize：核心池的大小
- * maximumPoolSize：最大线程数
- * keepAliveTime：线程没任务时最多保持多长时间后会终止

面试题：Java中多线程的创建有几种方式？四种。

java.lang.String类的使用

1. 概述

String: 字符串，使用一对""引起来表示。

1. **String**声明为final的，不可被继承

2. **String**实现了**Serializable**接口：表示字符串是支持序列化的。

实现了**Comparable**接口：表示**String**可以比较大小

3. **String**内部定义了final char[] value用于存储字符串数据

4. 通过字面量的方式（区别于new给一个字符串赋值，此时的字符串值声明在字符串常量池中）。

5. 字符串常量池中是不会存储相同内容(使用**String**类的**equals()**比较，返回true)的字符串的。

2. String的不可变性

2.1 说明

1. 当对字符串重新赋值时，需要重写指定内存区域赋值，不能使用原有的**value**进行赋值。

2. 当对现的字符串进行连接操作时，也需要重新指定内存区域赋值，不能使用原有的**value**进行赋值。

3. 当调用**String**的**replace()**方法修改指定字符或字符串时，也需要重新指定内存区域赋值，不能使用原有的**value**进行赋值。

2.2 代码举例

```
String s1 = "abc";//字面量的定义方式
String s2 = "abc";
s1 = "hello";

System.out.println(s1 == s2);//比较s1和s2的地址值

System.out.println(s1);//hello
System.out.println(s2)//abc

System.out.println("*****");

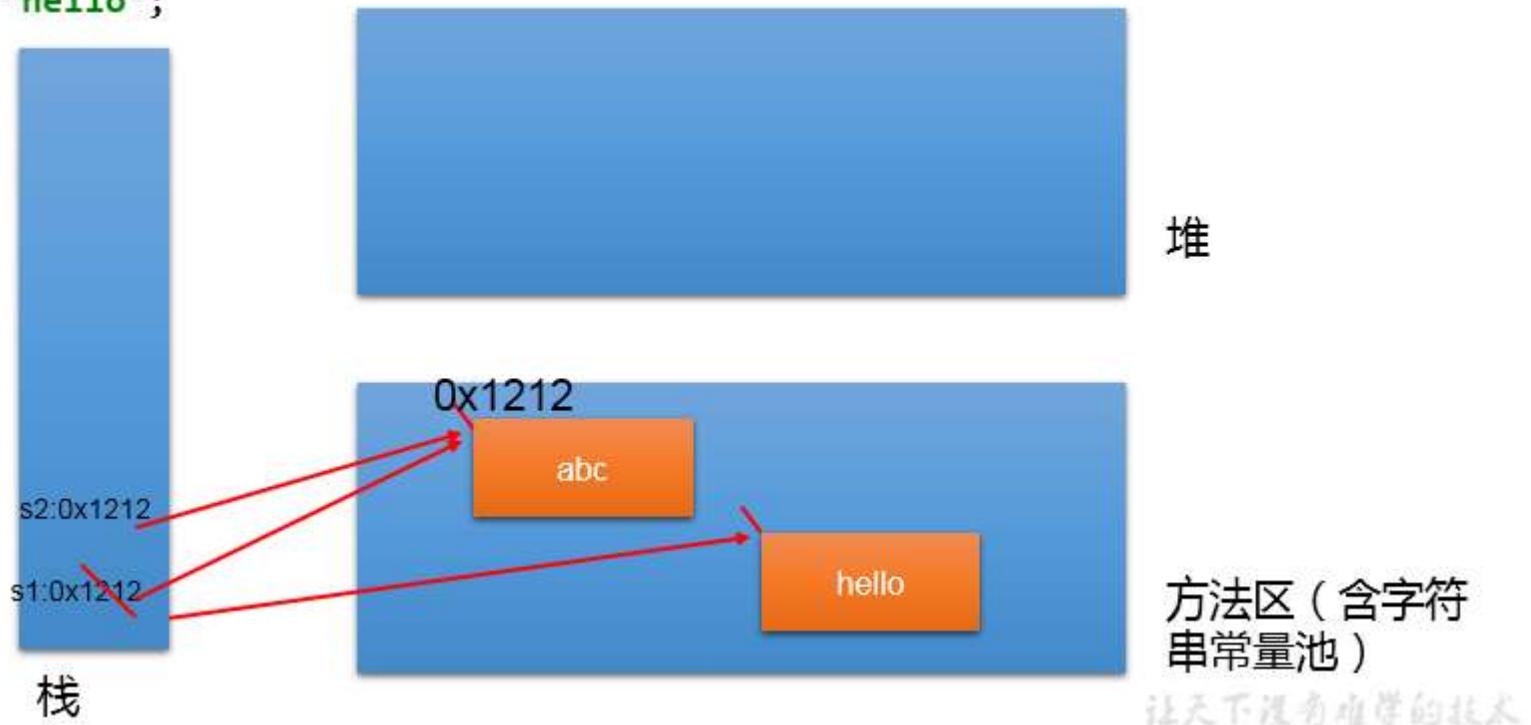
String s3 = "abc";
s3 += "def";
System.out.println(s3)//abcdef
System.out.println(s2);

System.out.println("*****");

String s4 = "abc";
String s5 = s4.replace('a', 'm');
System.out.println(s4)//abc
System.out.println(s5)//mbc
```

2.3 图示

```
String s1 = "abc"; //字面量的定义方式
String s2 = "abc";
s1 = "hello";
```



3.String实例化的方式

3.1 方式说明

方式一：通过字面量定义的方式

方式二：通过new + 构造器的方式

3.2 代码举例

//通过字面量定义的方式：此时的s1和s2的数据声明在方法区中的字符串常量池中。

```
String s1 = "javaEE";
String s2 = "javaEE";
```

//通过new + 构造器的方式：此时的s3和s4保存的地址值，是数据在堆空间中开辟空间以后对应的地址值。

```
String s3 = new String("javaEE");
String s4 = new String("javaEE");
```

```
System.out.println(s1 == s2); //true
System.out.println(s1 == s3); //false
System.out.println(s1 == s4); //false
System.out.println(s3 == s4); //false
```

3.3 面试题

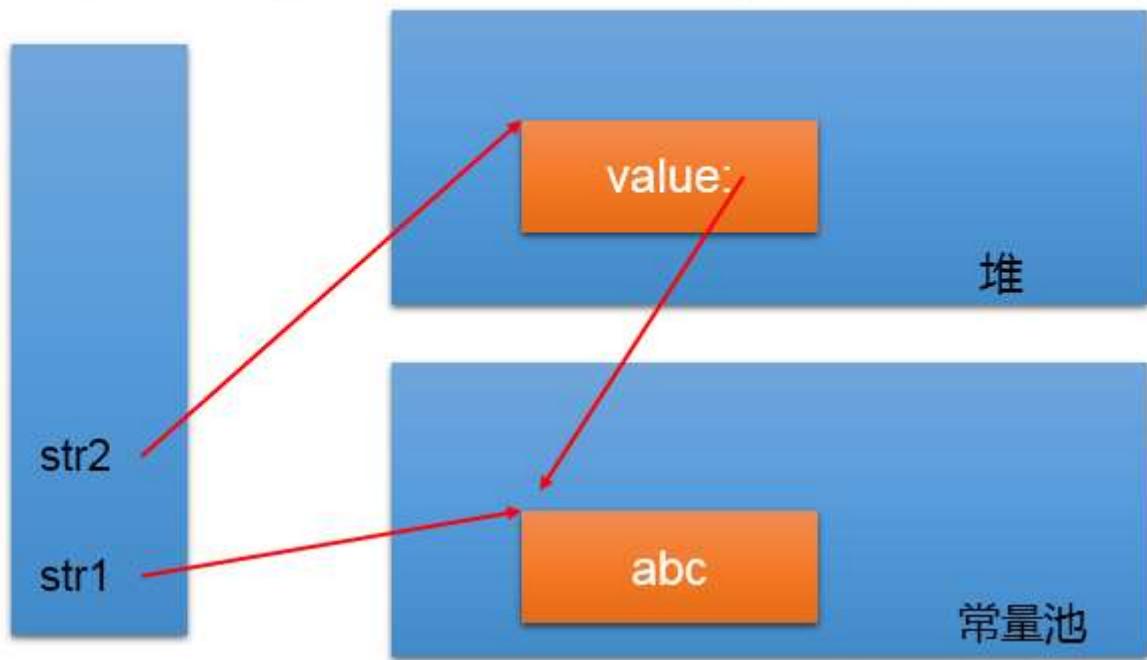
`String s = new String("abc");`方式创建对象，在内存中创建了几个对象？

两个：一个是堆空间中new结构，另一个是char[]对应的常量池中的数据：“abc”

3.4 图示

String str1 = "abc";与String str2 = new String("abc");的区别?

- 字符串常量存储在字符串常量池，目的是共享
- 字符串非常量对象存储在堆中。



4. 字符串拼接方式赋值的对比

4.1 说明

1. 常量与常量的拼接结果在常量池。且常量池中不会存在相同内容的常量。
2. 只要其中一个是变量，结果就在堆中。
3. 如果拼接的结果调用intern()方法，返回值就在常量池中

4.2 代码举例

```

String s1 = "javaEE";
String s2 = "hadoop";

String s3 = "javaEEhadoop";
String s4 = "javaEE" + "hadoop";
String s5 = s1 + "hadoop";
String s6 = "javaEE" + s2;
String s7 = s1 + s2;

System.out.println(s3 == s4); //true
System.out.println(s3 == s5); //false
System.out.println(s3 == s6); //false
System.out.println(s3 == s7); //false
System.out.println(s5 == s6); //false
System.out.println(s5 == s7); //false
System.out.println(s6 == s7); //false

String s8 = s6.intern(); //返回值得到的s8使用的常量值中已经存在的“javaEEhadoop”
System.out.println(s3 == s8); //true
*****



String s1 = "javaEEhadoop";
String s2 = "javaEE";
String s3 = s2 + "hadoop";
System.out.println(s1 == s3); //false

final String s4 = "javaEE"; //s4: 常量
String s5 = s4 + "hadoop";
System.out.println(s1 == s5); //true

```

5. 常用方法:

`int length(): 返回字符串的长度: return value.length`

`char charAt(int index):` 返回某索引处的字符`return value[index]`
`boolean isEmpty():` 判断是否是空字符串: `return value.length == 0`
`String toLowerCase():` 使用默认语言环境, 将 `String` 中的所字符转换为小写
`String toUpperCase():` 使用默认语言环境, 将 `String` 中的所字符转换为大写
`String trim():` 返回字符串的副本, 忽略前导空白和尾部空白
`boolean equals(Object obj):` 比较字符串的内容是否相同
`boolean equalsIgnoreCase(String anotherString):` 与 `equals` 方法类似, 忽略大小写
`String concat(String str):` 将指定字符串连接到此字符串的结尾。等价于用“+”
`int compareTo(String anotherString):` 比较两个字符串的大小
`String substring(int beginIndex):` 返回一个新的字符串, 它是此字符串的从 `beginIndex` 开始截取到最后的一个子字符串。
`String substring(int beginIndex, int endIndex):` 返回一个新字符串, 它是此字符串从 `beginIndex` 开始截取到 `endIndex` (不包含) 的一个子字符串。
`boolean endsWith(String suffix):` 测试此字符串是否以指定的后缀结束
`boolean startsWith(String prefix):` 测试此字符串是否以指定的前缀开始
`boolean startsWith(String prefix, int toffset):` 测试此字符串从指定索引开始的子字符串是否以指定前缀开始
`boolean contains(CharSequence s):` 当且仅当此字符串包含指定的 `char` 值序列时, 返回 `true`
`int indexOf(String str):` 返回指定子字符串在此字符串中第一次出现处的索引
`int indexOf(String str, int fromIndex):` 返回指定子字符串在此字符串中第一次出现处的索引, 从指定的索引开始
`int lastIndexOf(String str):` 返回指定子字符串在此字符串中最右边出现处的索引
`int lastIndexOf(String str, int fromIndex):` 返回指定子字符串在此字符串中最后一次出现处的索引, 从指定的索引开始反向搜索

注: `indexOf` 和 `lastIndexOf` 方法如果未找到都是返回-1

替换:

`String replace(char oldChar, char newChar):` 返回一个新的字符串, 它是通过用 `newChar` 替换此字符串中出现的所 `oldChar` 得到的。
`String replace(CharSequence target, CharSequence replacement):` 使用指定的字面值替换序列替换此字符串所匹配字面值目标序列的子字符串。
`String replaceAll(String regex, String replacement):` 使用给定的 `replacement` 替换此字符串所匹配给定的正则表达式的子字符串。
`String replaceFirst(String regex, String replacement):` 使用给定的 `replacement` 替换此字符串匹配给定的正则表达式的一个子字符串。

匹配:

`boolean matches(String regex):` 告知此字符串是否匹配给定的正则表达式。

切片:

`String[] split(String regex):` 根据给定正则表达式的匹配拆分此字符串。
`String[] split(String regex, int limit):` 根据匹配给定的正则表达式来拆分此字符串, 最多不超过 `limit` 个, 如果超过了, 剩下的全部都放到最后一个元素中。

6. String与其它结构的转换

6.1 与基本数据类型、包装类之间的转换

`String --> 基本数据类型、包装类:` 调用包装类的静态方法: `parseXxx(str)`
`基本数据类型、包装类 --> String:` 调用 `String` 重载的 `valueOf(xxx)`

```

@Test
public void test1(){
    String str1 = "123";
//    int num = (int)str1;//错误的
    int num = Integer.parseInt(str1);

    String str2 = String.valueOf(num); // "123"
    String str3 = num + "";
}

```

```

        System.out.println(str1 == str3);
    }

```

6.2 与字符数组之间的转换

String --> char[]: 调用String的toCharArray()
char[] --> String: 调用String的构造器

```

@Test
public void test2(){
    String str1 = "abc123"; //题目: a21cb3

    char[] charArray = str1.toCharArray();
    for (int i = 0; i < charArray.length; i++) {
        System.out.println(charArray[i]);
    }

    char[] arr = new char[]{'h','e','l','l','o'};
    String str2 = new String(arr);
    System.out.println(str2);
}

```

6.3 与字节数组之间的转换

编码: String --> byte[]: 调用String的getBytes()
解码: byte[] --> String: 调用String的构造器

编码: 字符串 -->字节 (看得懂 --->看不懂的二进制数据)
解码: 编码的逆过程, 字节 --> 字符串 (看不懂的二进制数据 ---> 看得懂)

说明: 解码时, 要求解码使用的字符集必须与编码时使用的字符集一致, 否则会出现乱码。

```

@Test
public void test3() throws UnsupportedEncodingException {
    String str1 = "abc123中国";
    byte[] bytes = str1.getBytes(); //使用默认的字符集, 进行编码。
    System.out.println(Arrays.toString(bytes));

    byte[] gbks = str1.getBytes("gbk"); //使用gbk字符集进行编码。
    System.out.println(Arrays.toString(gbks));

    System.out.println("*****");
    String str2 = new String(bytes); //使用默认的字符集, 进行解码。
    System.out.println(str2);

    String str3 = new String(gbks);
    System.out.println(str3); //出现乱码。原因: 编码集和解码集不一致!

    String str4 = new String(gbks, "gbk");
    System.out.println(str4); //没出现乱码。原因: 编码集和解码集一致!
}

```

6.4 与StringBuffer、StringBuilder之间的转换

String -->StringBuffer、StringBuilder: 调用StringBuffer、StringBuilder构造器

StringBuffer、StringBuilder -->String: ①调用String构造器; ②StringBuffer、StringBuilder的

`toString()`

7. JVM中字符串常量池存放位置说明：

jdk 1.6 (jdk 6.0 ,java 6.0):字符串常量池存储在方法区（永久区）

jdk 1.7:字符串常量池存储在堆空间

jdk 1.8:字符串常量池存储在方法区（元空间）

8. 常见算法题目的考查：

1) 模拟一个**trim**方法，去除字符串两端的空格。

2) 将一个字符串进行反转。将字符串中指定部分进行反转。比如“**abcdefg**”反转为“**abfedcg**”

3) 获取一个字符串在另一个字符串中出现的次数。

比如：获取“**ab**”在“**abkkcadkabkebfkabkskab**”中出现的次数

4) 获取两个字符串中最大相同子串。比如：

`str1 = "abcwerthelloyuiodef";str2 = "cvhellobnm"`

提示：将短的那个串进行长度依次递减的子串与较长的串比较。

5) 对字符串中字符进行自然顺序排序。

提示：

1字符串变成字符数组。

2对数组排序，择，冒泡，`Arrays.sort()`;

3将排序后的数组变成字符串。

1. String、StringBuffer、StringBuilder三者的对比

String: 不可变的字符序列；底层使用char[]存储

StringBuffer: 可变的字符序列；线程安全的，效率低；底层使用char[]存储

StringBuilder: 可变的字符序列；jdk5.0新增的，线程不安全的，效率高；底层使用char[]存储

2. StringBuffer与StringBuilder的内存解析

以StringBuffer为例：

```
String str = new String(); //char[] value = new char[0];
String str1 = new String("abc"); //char[] value = new char[]{'a', 'b', 'c'};

StringBuffer sb1 = new StringBuffer(); //char[] value = new char[16]; 底层创建了一个长度是16的数组。
System.out.println(sb1.length()); //
sb1.append('a'); //value[0] = 'a';
sb1.append('b'); //value[1] = 'b';

StringBuffer sb2 = new StringBuffer("abc"); //char[] value = new char["abc".length() + 16];

//问题1. System.out.println(sb2.length()); //3
//问题2. 扩容问题：如果要添加的数据底层数组盛不下了，那就需要扩容底层的数组。
    默认情况下，扩容为原来容量的2倍 + 2，同时将原数组中的元素复制到新的数组中。
```

指导意义：开发中建议大家使用：StringBuffer(int capacity) 或 StringBuilder(int capacity)

3. 对比String、StringBuffer、StringBuilder三者的执行效率

从高到低排列：StringBuilder > StringBuffer > String

4. StringBuffer、StringBuilder中的常用方法

- 增：append(xxx)
- 删：delete(int start, int end)
- 改：setCharAt(int n, char ch) / replace(int start, int end, String str)
- 查：charAt(int n)
- 插：insert(int offset, xxx)
- 长度：length();
- *遍历：for() + charAt() / toString()

1. 获取系统当前时间: *System*类中的`currentTimeMillis()`

```
long time = System.currentTimeMillis();
//返回当前时间与1970年1月1日0时0分0秒之间以毫秒为单位的时间差。
//称为时间戳
System.out.println(time);
```

2. `java.util.Date`类与`java.sql.Date`类

```
/*
 * java.util.Date类
 * |---java.sql.Date类
 *
 * 1. 两个构造器的使用
 *    >构造器一: Date(): 创建一个对应当前时间的Date对象
 *    >构造器二: 创建指定毫秒数的Date对象
 *
 * 2. 两个方法的使用
 *    >toString(): 显示当前的年、月、日、时、分、秒
 *    >getTime(): 获取当前Date对象对应的毫秒数。 (时间戳)
 *
 * 3. java.sql.Date对应着数据库中的日期类型的变量
 *    >如何实例化
 *    >如何将java.util.Date对象转换为java.sql.Date对象
 */
@Test
public void test2(){
    //构造器一: Date(): 创建一个对应当前时间的Date对象
    Date date1 = new Date();
    System.out.println(date1.toString()); //Sat Feb 16 16:35:31 GMT+08:00
2019

    System.out.println(date1.getTime()); //1550306204104

    //构造器二: 创建指定毫秒数的Date对象
    Date date2 = new Date(155030620410L);
    System.out.println(date2.toString());

    //创建java.sql.Date对象
    java.sql.Date date3 = new java.sql.Date(35235325345L);
    System.out.println(date3); //1971-02-13

    //如何将java.util.Date对象转换为java.sql.Date对象
    //情况一:
    //    Date date4 = new java.sql.Date(2343243242323L);
    //    java.sql.Date date5 = (java.sql.Date) date4;
    //情况二:
    Date date6 = new Date();
    java.sql.Date date7 = new java.sql.Date(date6.getTime());
}

}
```

3. `java.text.SimpleDateFormat`类

`SimpleDateFormat`对日期`Date`类的格式化和解析

1. 两个操作:

1.1 格式化: 日期 ---> 字符串

1.2 解析: 格式化的逆过程, 字符串 ---> 日期

2. `SimpleDateFormat`的实例化:`new + 构造器`

```

//*****照指定的方式格式化和解析: 调用带参的构造器*****
//      SimpleDateFormat sdf1 = new SimpleDateFormat("yyyy.MMMMd.dd
GGG hh:mm aaa");
      SimpleDateFormat sdf1 = new SimpleDateFormat("yyyy-MM-dd
hh:mm:ss");
      //格式化
String format1 = sdf1.format(date);
System.out.println(format1); //2019-02-18 11:48:27
      //解析: 要求字符串必须是符合SimpleDateFormat识别的格式(通过构造器参数体现),
      //否则, 抛异常
Date date2 = sdf1.parse("2020-02-18 11:48:27");
System.out.println(date2);

```

小练习:

```

/*
    练习一: 字符串"2020-09-08"转换为java.sql.Date

    练习二: "天打渔两天晒网"   1990-01-01  xxxx-xx-xx 打渔? 晒网?

    举例: 2020-09-08 ? 总天数

    总天数 % 5 == 1,2,3 : 打渔
    总天数 % 5 == 4,0 : 晒网

    总天数的计算?
    方式一: ( date2.getTime() - date1.getTime() ) / (1000 * 60 * 60 * 24) +
1
    方式二: 1990-01-01 --> 2019-12-31 + 2020-01-01 --> 2020-09-08
*/
@Test
public void testExer() throws ParseException {
    String birth = "2020-09-08";

    SimpleDateFormat sdf1 = new SimpleDateFormat("yyyy-MM-dd");
    Date date = sdf1.parse(birth);
    System.out.println(date);

    java.sql.Date birthDate = new java.sql.Date(date.getTime());
    System.out.println(birthDate);
}

```

4.Calendar类: 日历类、抽象类

```

//1. 实例化
//方式一: 创建其子类(GregorianCalendar)的对象
//方式二: 调用其静态方法getInstance()
Calendar calendar = Calendar.getInstance();
System.out.println(calendar.getClass());

//2. 常用方法
//get()
int days = calendar.get(Calendar.DAY_OF_MONTH);
System.out.println(days);
System.out.println(calendar.get(Calendar.DAY_OF_YEAR));

//set()
//calendar可变性
calendar.set(Calendar.DAY_OF_MONTH, 22);
days = calendar.get(Calendar.DAY_OF_MONTH);
System.out.println(days);

//add()

```

```
calendar.add(Calendar.DAY_OF_MONTH,-3);
days = calendar.get(Calendar.DAY_OF_MONTH);
System.out.println(days);

//getTime():日历类---> Date
Date date = calendar.getTime();
System.out.println(date);

//setTime():Date ---> 日历类
Date date1 = new Date();
calendar.setTime(date1);
days = calendar.get(Calendar.DAY_OF_MONTH);
System.out.println(days);
```

1. 日期时间API的迭代:

第一代: jdk 1.0 Date类

第二代: jdk 1.1 Calendar类, 一定程度上替换Date类

第三代: jdk 1.8 提出了新的一套API

2. 前两代存在的问题举例:

可变性: 像日期和时间这样的类应该是不可变的。

偏移性: Date中的年份是从1900开始的, 而月份都从0开始。

格式化: 格式化只对Date用, Calendar则不行。

此外, 它们也不是线程安全的; 不能处理闰秒等。

3. java 8 中新的日期时间API涉及到的包

java.time – 包含值对象的基础包

java.time.chrono – 提供对不同的日历系统的访问

java.time.format – 格式化和解析时间和日期

java.time.temporal – 包括底层框架和扩展特性

java.time.zone – 包含时区支持的类

说明: 大多数开发者只会用到基础包和format包, 也可能会用到temporal包。因此, 尽管有68个新的公开类型, 大多数开发者, 大概将只会用到其中的三分之一。

4. 本地日期、本地时间、本地日期时间的使用: LocalDate / LocalTime / LocalDateTime

4.1 说明:

① 分别表示使用 ISO-8601日历系统的日期、时间、日期和时间。它们提供了简单的本地日期或时间, 并不包含当前的时间信息, 也不包含与时区相关的信息。

② *LocalDateTime*相较于*LocalDate*、*LocalTime*, 使用频率要高

③ 类似于*Calendar*

4.2 常用方法:

方法	描述
<code>now() /* now(ZonedDateTime zone)</code>	静态方法，根据当前时间创建对象/指定时区的对象
<code>of()</code>	静态方法，根据指定日期/时间创建对象
<code>getDayOfMonth()/getDayOfYear()</code>	获得月份天数(1-31) /获得年份天数(1-366)
<code>getDayOfWeek()</code>	获得星期几(返回一个 DayOfWeek 枚举值)
<code>getMonth()</code>	获得月份, 返回一个 Month 枚举值
<code>getMonthValue() / getYear()</code>	获得月份(1-12) /获得年份
<code>getHour()/getMinute()/getSecond()</code>	获得当前对象对应的小时、分钟、秒
<code>withDayOfMonth()/withDayOfYear()/withMonth()/withYear()</code>	将月份天数、年份天数、月份、年份修改为指定的值并返回新的对象
<code>plusDays(), plusWeeks(), plusMonths(), plusYears(),plusHours()</code>	向当前对象添加几天、几周、几个月、几年、几小时
<code>minusMonths() / minusWeeks() / minusDays()/minusYears()/minusHours()</code>	从当前对象减去几月、几周、几天、几年、几小时

5. 时间点: Instant

5.1 说明:

- ① 时间线上的一个瞬时点。 概念上讲，它只是简单的表示自1970年1月1日0时0分0秒 (UTC开始的秒数。)
- ② 类似于 `java.util.Date`类

5.2 常用方法:

方法	描述
<code>now()</code>	静态方法，返回默认UTC时区的Instant类的对象
<code>ofEpochMilli(long epochMilli)</code>	静态方法，返回在1970-01-01 00:00:00基础上加上指定毫秒数之后的Instant类的对象
<code>atOffset(ZoneOffset offset)</code>	结合即时的偏移来创建一个 OffsetDateTime
<code>toEpochMilli()</code>	返回1970-01-01 00:00:00到当前时间的毫秒数，即为时间戳

时间戳是指格林威治时间1970年01月01日00时00分00秒(北京时间1970年01月01日08时00分00秒)起至现在的总秒数。

让天下没有难学的IT

6. 日期时间格式化类: DateTimeFormatter

6.1 说明:

- ① 格式化或解析日期、时间
- ② 类似于 `SimpleDateFormat`

6.2 常用方法:

① 实例化方式:

- 预定义的标准格式。如: `ISO_LOCAL_DATE_TIME;ISO_LOCAL_DATE;ISO_LOCAL_TIME`
- 本地化相关的格式。如: `ofLocalizedDateTime(FormatStyle.LONG)`

- 自定义的格式。如: `ofPattern("yyyy-MM-dd hh:mm:ss")`

- ⊙ 常用方法:

方法	描述
<code>ofPattern(String pattern)</code>	静态方法，返回一个指定字符串格式的 <code>DateTimeFormatter</code>
<code>format(TemporalAccessor t)</code>	格式化一个日期、时间，返回字符串
<code>parse(CharSequence text)</code>	将指定格式的字符序列解析为一个日期、时间

- 特别的：自定义的格式。如: `ofPattern("yyyy-MM-dd hh:mm:ss")`

```
// 重点: 自定义的格式。如: ofPattern("yyyy-MM-dd hh:mm:ss")
DateTimeFormatter formatter3 = DateTimeFormatter.ofPattern("yyyy-MM-dd
hh:mm:ss");
// 格式化
String str4 = formatter3.format(LocalDateTime.now());
System.out.println(str4); // 2019-02-18 03:52:09
```

```
// 解析
TemporalAccessor accessor = formatter3.parse("2019-02-18 03:52:09");
System.out.println(accessor);
```

7. 其它API的使用（不讲）

7.1 带时区的日期时间: `ZonedDateTime / ZoneId`

举例：

```
// ZoneId: 类中包含了所有的时区信息
@Test
public void test1(){
    // getAvailableZoneIds(): 获取所有的ZoneId
    Set<String> zoneIds = ZoneId.getAvailableZoneIds();
    for(String s : zoneIds){
        System.out.println(s);
    }
    System.out.println();

    // 获取“Asia/Tokyo”时区对应的时间
    LocalDateTime localDateTime = LocalDateTime.now(ZoneId.of("Asia/
Tokyo"));
    System.out.println(localDateTime);

}

// ZonedDateTime: 带时区的日期时间
@Test
public void test2(){
    // now(): 获取本时区的ZonedDateTime对象
    ZonedDateTime zonedDateTime = ZonedDateTime.now();
    System.out.println(zonedDateTime);
    // now(ZoneId id): 获取指定时区的ZonedDateTime对象
    ZonedDateTime zonedDateTime1 = ZonedDateTime.now(ZoneId.of("Asia/
Tokyo"));
    System.out.println(zonedDateTime1);
}
```

7.2 时间间隔: Duration--用于计算两个“时间”间隔, 以秒和纳秒为基准

方法	描述
between(Temporal start,Temporal end)	静态方法, 返回Duration对象, 表示两个时间的间隔
getNano()/getSeconds()	返回时间间隔的纳秒数/返回时间间隔的秒数
toDays()/toHours()/toMinutes()/ toMillis()/toNanos()	返回时间间隔期间的天数、小时数、分钟数、毫秒数、纳秒数

举例:

```

@Test
public void test3(){
    LocalTime localTime = LocalTime.now();
    LocalTime localTime1 = LocalTime.of(15, 23, 32);
    //between():静态方法, 返回Duration对象, 表示两个时间的间隔
    Duration duration = Duration.between(localTime1, localTime);
    System.out.println(duration);

    System.out.println(duration.getSeconds());
    System.out.println(duration.getNano());

    LocalDateTime localDateTime = LocalDateTime.of(2016, 6, 12, 15, 23,
32);
    LocalDateTime localDateTime1 = LocalDateTime.of(2017, 6, 12, 15, 23,
32);

    Duration duration1 = Duration.between(localDateTime1, localDateTime);
    System.out.println(duration1.toDays());

}

```

7.3 日期间隔: Period --用于计算两个“日期”间隔, 以年、月、日衡量

方法	描述
between(LocalDate start,LocalDate end)	静态方法, 返回Period对象, 表示两个本地日期的间隔
getYears()/getMonths()/getDays()	返回此期间的年数、月数、天数
withYears(int years)/withMonths(int months)/withDays(int days)	返回设置间隔指定年、月、日数以后的Period对象

举例:

```

@Test
public void test4(){
    LocalDate localDate = LocalDate.now();
    LocalDate localDate1 = LocalDate.of(2028, 3, 18);

    Period period = Period.between(localDate, localDate1);
    System.out.println(period);

    System.out.println(period.getYears());
    System.out.println(period.getMonths());
    System.out.println(period.getDays());

    Period period1 = period.withYears(2);
    System.out.println(period1);

}

```

7.4 日期时间校正器：TemporalAdjuster

举例：

```
@Test
public void test5(){
    //获取当前日期的下一个周日是哪天？
    TemporalAdjuster temporalAdjuster = TemporalAdjusters.next
(DayOfWeek.SUNDAY);

    LocalDateTime localDateTime = LocalDateTime.now().with
(temporalAdjuster);
    System.out.println(localDateTime);

    //获取下一个工作日是哪天？
    LocalDate localDate = LocalDate.now().with(new TemporalAdjuster(){

        @Override
        public Temporal adjustInto(Temporal temporal) {
            LocalDate date = (LocalDate)temporal;
            if(date.getDayOfWeek().equals(DayOfWeek.FRIDAY)){
                return date.plusDays(3);
            }else if(date.getDayOfWeek().equals(DayOfWeek.SATURDAY)){
                return date.plusDays(2);
            }else{
                return date.plusDays(1);
            }
        }
    });

    System.out.println("下一个工作日是：" + localDate);
}
```

1. Java比较器的使用背景：

Java中的对象，正常情况下，只能进行比较：== 或 !=。不能使用 > 或 < 的。但是在开发场景中，我们需要对多个对象进行排序，言外之意，就需要比较对象的大小。如何实现？使用两个接口中的任何一个：Comparable 或 Comparator

2. 自然排序：使用Comparable接口

2.1 说明

1. 像String、包装类等实现了Comparable接口，重写了compareTo(obj)方法，给出了比较两个对象大小的方式。

2. 像String、包装类重写compareTo()方法以后，进行了从小到大的排列

3. 重写compareTo(obj)的规则：

如果当前对象this大于形参对象obj，则返回正整数，

如果当前对象this小于形参对象obj，则返回负整数，

如果当前对象this等于形参对象obj，则返回零。

4. 对于自定义类来说，如果需要排序，我们可以让自定义类实现Comparable接口，重写compareTo(obj)方法。在compareTo(obj)方法中指明如何排序

2.2 自定义类代码举例：

```
public class Goods implements Comparable{

    private String name;
    private double price;

    //指明商品比较大小的方式：照价格从低到高排序，再照产品名称从高到低排序
    @Override
    public int compareTo(Object o) {
        // System.out.println("*****");
        if(o instanceof Goods){
            Goods goods = (Goods)o;
            //方式一：
            if(this.price > goods.price){
                return 1;
            }else if(this.price < goods.price){
                return -1;
            }else{
                return 0;
            }
            //方式二：
            return Double.compare(this.price,goods.price);
        }
        // return 0;
        throw new RuntimeException("传入的数据类型不一致！");
    }
    // getter、setter、toString()、构造器：省略
}
```

3. 定制排序：使用Comparator接口

3.1 说明

1. 背景：

当元素的类型没实现java.lang.Comparable接口而又不方便修改代码，或者实现了java.lang.Comparable接口的排序规则不适合当前的操作，那么可以考虑使用Comparator的对象来排序

2. 重写compare(Object o1, Object o2)方法，比较o1和o2的大小：

如果方法返回正整数，则表示o1大于o2；

如果返回0，表示相等；

返回负整数，表示o1小于o2。

3.2 代码举例:

```

Comparator com = new Comparator() {
    //指明商品比较大小的方式:照产品名称从低到高排序,再照价格从高到低排序
    @Override
    public int compare(Object o1, Object o2) {
        if(o1 instanceof Goods && o2 instanceof Goods){
            Goods g1 = (Goods)o1;
            Goods g2 = (Goods)o2;
            if(g1.getName().equals(g2.getName())){
                return -Double.compare(g1.getPrice(),g2.getPrice());
            }else{
                return g1.getName().compareTo(g2.getName());
            }
        }
        throw new RuntimeException("输入的数据类型不一致");
    }
}

```

使用:

```

Arrays.sort(goods,com);
Collections.sort(coll,com);
new TreeSet(com);

```

4. 两种排序方式对比

- * Comparable接口的方式一旦一定, 保证Comparable接口实现类的对象在任何位置都可以比较大小。
- * Comparator接口属于临时性的比较。

其他类

1. System类

- System类代表系统，系统级的很多属性和控制方法都放置在该类的内部。该类位于java.lang包。
- 由于该类的构造器是private的，所以无法创建该类的对象，也就是无法实例化该类。其内部的成员变量和成员方法都是static的，所以也可以很方便的进行调用。
- 方法：
 - ⌚ native long currentTimeMillis()
 - ⌚ void exit(int status)
 - ⌚ void gc()
 - ⌚ String getProperty(String key)

2. Math类

java.lang.Math提供了一系列静态方法用于科学计算。其方法的参数和返回值类型一般为double型。

3. BigInteger类、 BigDecimal类

说明：

- ① java.math包的**BigInteger**可以表示不可变的任意精度的整数。
- ② 要求数字精度比较高，用到**java.math.BigDecimal**类

代码举例：

```
public void testBigInteger() {  
    BigInteger bi = new BigInteger("12433241123");  
    BigDecimal bd = new BigDecimal("12435.351");  
    BigDecimal bd2 = new BigDecimal("11");  
    System.out.println(bi);  
    // System.out.println(bd.divide(bd2));  
    System.out.println(bd.divide(bd2, BigDecimal.ROUND_HALF_UP));  
    System.out.println(bd.divide(bd2, 15, BigDecimal.ROUND_HALF_UP));  
}
```

1. 枚举类的说明:

- * 1. 枚举类的理解: 类的对象只有有限个, 确定的。我们称此类为枚举类
- * 2. 当需要定义一组常量时, 强烈建议使用枚举类
- * 3. 如果枚举类中只有一个对象, 则可以作为单例模式的实现方式。

2. 如何自定义枚举类? 步骤:

```
//自定义枚举类
class Season{
    //1. 声明Season对象的属性:private final修饰
    private final String seasonName;
    private final String seasonDesc;

    //2. 私化类的构造器, 并给对象属性赋值
    private Season(String seasonName, String seasonDesc){
        this.seasonName = seasonName;
        this.seasonDesc = seasonDesc;
    }

    //3. 提供当前枚举类的多个对象: public static final的
    public static final Season SPRING = new Season("春天", "春暖花开");
    public static final Season SUMMER = new Season("夏天", "夏日炎炎");
    public static final Season AUTUMN = new Season("秋天", "秋高气爽");
    public static final Season WINTER = new Season("冬天", "冰天雪地");

    //4. 其他诉求1: 获取枚举类对象的属性
    public String getSeasonName() {
        return seasonName;
    }

    public String getSeasonDesc() {
        return seasonDesc;
    }

    //4. 其他诉求1: 提供toString()
    @Override
    public String toString() {
        return "Season{" +
            "seasonName='" + seasonName + '\'' +
            ", seasonDesc='" + seasonDesc + '\'' +
            '}';
    }
}
```

3. jdk 5.0 新增使用enum定义枚举类。步骤:

```
//使用enum关键字枚举类
enum Season1 {
    //1. 提供当前枚举类的对象, 多个对象之间用", "隔开, 末尾对象"; "结束
    SPRING("春天", "春暖花开"),
    SUMMER("夏天", "夏日炎炎"),
    AUTUMN("秋天", "秋高气爽"),
    WINTER("冬天", "冰天雪地");

    //2. 声明Season对象的属性:private final修饰
    private final String seasonName;
    private final String seasonDesc;

    //2. 私化类的构造器, 并给对象属性赋值
}
```

```

private Season1(String seasonName, String seasonDesc) {
    this.seasonName = seasonName;
    this.seasonDesc = seasonDesc;
}

//4. 其他诉求1： 获取枚举类对象的属性
public String getSeasonName() {
    return seasonName;
}

public String getSeasonDesc() {
    return seasonDesc;
}

}

```

4. 使用enum定义枚举类之后，枚举类常用方法：（继承于java.lang.Enum类）

```

Season1 summer = Season1.SUMMER;
//toString(): 返回枚举类对象的名称
System.out.println(summer.toString());

//      System.out.println(Season1.class.getSuperclass());
System.out.println("*****");
//values(): 返回所有的枚举类对象构成的数组
Season1[] values = Season1.values();
for(int i = 0; i < values.length; i++) {
    System.out.println(values[i]);
}
System.out.println("*****");
Thread.State[] values1 = Thread.State.values();
for (int i = 0; i < values1.length; i++) {
    System.out.println(values1[i]);
}

//valueOf(String objName): 返回枚举类中对象名是objName的对象。
Season1 winter = Season1.valueOf("WINTER");
//如果没objName的枚举类对象，则抛异常：IllegalArgumentException
//      Season1 winter = Season1.valueOf("WINTER1");
System.out.println(winter);

```

5. 使用enum定义枚举类之后，如何让枚举类对象分别实现接口：

```

interface Info{
    void show();
}

//使用enum关键字枚举类
enum Season1 implements Info{
    //1. 提供当前枚举类的对象，多个对象之间用","隔开，末尾对象";"结束
    SPRING("春天", "春暖花开"){
        @Override
        public void show() {
            System.out.println("春天在哪里？");
        }
    },
    SUMMER("夏天", "夏日炎炎"){
        @Override
        public void show() {
            System.out.println("宁夏");
        }
    }
}

```

```
    }
},
AUTUMN("秋天", "秋高气爽"){
    @Override
    public void show() {
        System.out.println("秋天不回来");
    }
},
WINTER("冬天", "冰天雪地"){
    @Override
    public void show() {
        System.out.println("大约在冬季");
    }
}
};
```

1. 注解的理解

- ① **jdk 5.0 新增的功能**
- *
- ② **Annotation** 其实就是代码里的特殊标记，这些标记可以在编译，类加载，运行时被读取，并执行相应的处理。通过使用 **Annotation**，
- * 程序员可以在不改变原逻辑的情况下，在源文件中嵌入一些补充信息。
- *
- ③ 在 JavaSE 中，注解的使用目的比较简单，例如标记过时的功能，忽略警告等。在 JavaEE / Android
- * 中注解占据了更重要的角色，例如用来配置应用程序的任何切面，代替 JavaEE 旧版中所遗留的繁冗
- * 代码和 XML 配置等。

框架 = 注解 + 反射机制 + 设计模式

2. 注解的使用示例

- * **示例一：生成文档相关的注解**
- * **示例二：在编译时进行格式检查(JDK 内置的基本注解)**
 - @Override**: 限定重写父类方法，该注解只能用于方法
 - @Deprecated**: 用于表示所修饰的元素(类，方法等)已过时。通常是因为所修饰的结构危险或存在更好的择
 - @SuppressWarnings**: 抑制编译器警告
- * **示例：跟踪代码依赖性，实现替代配置文件功能**

3. 如何自定义注解：参照 **@SuppressWarnings** 定义

- * ① 注解声明为： **@interface**
- * ② 内部定义成员，通常使用 **value** 表示
- * ③ 可以指定成员的默认值，使用 **default** 定义
- * ④ 如果自定义注解没成员，表明是一个标识作用。

说明：

如果注解有成员，在使用注解时，需要指明成员的值。
自定义注解必须配上注解的信息处理流程(使用反射)才意义。
自定义注解通过都会指明两个元注解：**Retention**、**Target**

代码举例：

```

@Inherited
@Repeatable(MyAnnotations.class)
@Retention(RetentionPolicy.RUNTIME)
@Target({TYPE, FIELD, METHOD, PARAMETER, CONSTRUCTOR,
LOCAL_VARIABLE, TYPE_PARAMETER, TYPE_USE})
public @interface MyAnnotation {

    String value() default "hello";
}

```

4. 元注解：对现有的注解进行解释说明的注解。

jdk 提供的4种元注解：
Retention: 指定所修饰的 **Annotation** 的生命周期: **SOURCE\CLASS** (默认行为\RUNTIME)
只声明为 **RUNTIME** 生命周期的注解，才能通过反射获取。

Target: 用于指定被修饰的 *Annotation* 能用于修饰哪些程序元素

*****出现的频率较低*****

Documented: 表示所修饰的注解在被 *javadoc* 解析时，保留下来。

Inherited: 被它修饰的 *Annotation* 将具继承性。

--->类比：元数据的概念：String name = "Tom";

5. 如何获取注解信息：通过发射来进行获取、调用。

前提：要求此注解的元注解 *Retention* 中声明的生命周期状态为：*RUNTIME*。

6. JDK8中注解的新特性：可重复注解、类型注解

6.1 可重复注解：① 在 *MyAnnotation* 上声明 *@Repeatable*，成员值为

MyAnnotations.class

② *MyAnnotation* 的 *Target* 和 *Retention* 等元注解与 *MyAnnotations* 相同。

6.2 类型注解：

ElementType.TYPE_PARAMETER 表示该注解能写在类型变量的声明语句中（如：泛型声明）。

ElementType.TYPE_USE 表示该注解能写在使用类型的任何语句中。

1. 集合与数组存储数据概述:

集合、数组都是对多个数据进行存储操作的结构，简称Java容器。

说明：此时的存储，主要指的是内存层面的存储，不涉及到持久化的存储
(.txt,.jpg,.avi, 数据库中)

2. 数组存储的特点：

- > 一旦初始化以后，其长度就确定了。
- > 数组一旦定义好，其元素的类型也就确定了。我们也就只能操作指定类型的数据了。
* 比如：`String[] arr; int[] arr1; Object[] arr2;`

3. 数组存储的弊端：

- * > 一旦初始化以后，其长度就不可修改。
- * > 数组中提供的方法非常限，对于添加、删除、插入数据等操作，非常不便，同时效率不高。
- * > 获取数组中实际元素的个数的需求，数组没有现成的属性或方法可用
- * > 数组存储数据的特点：有序、可重复。对于无序、不可重复的需求，不能满足。

4. 集合存储的优点：

解决数组存储数据方面的弊端。

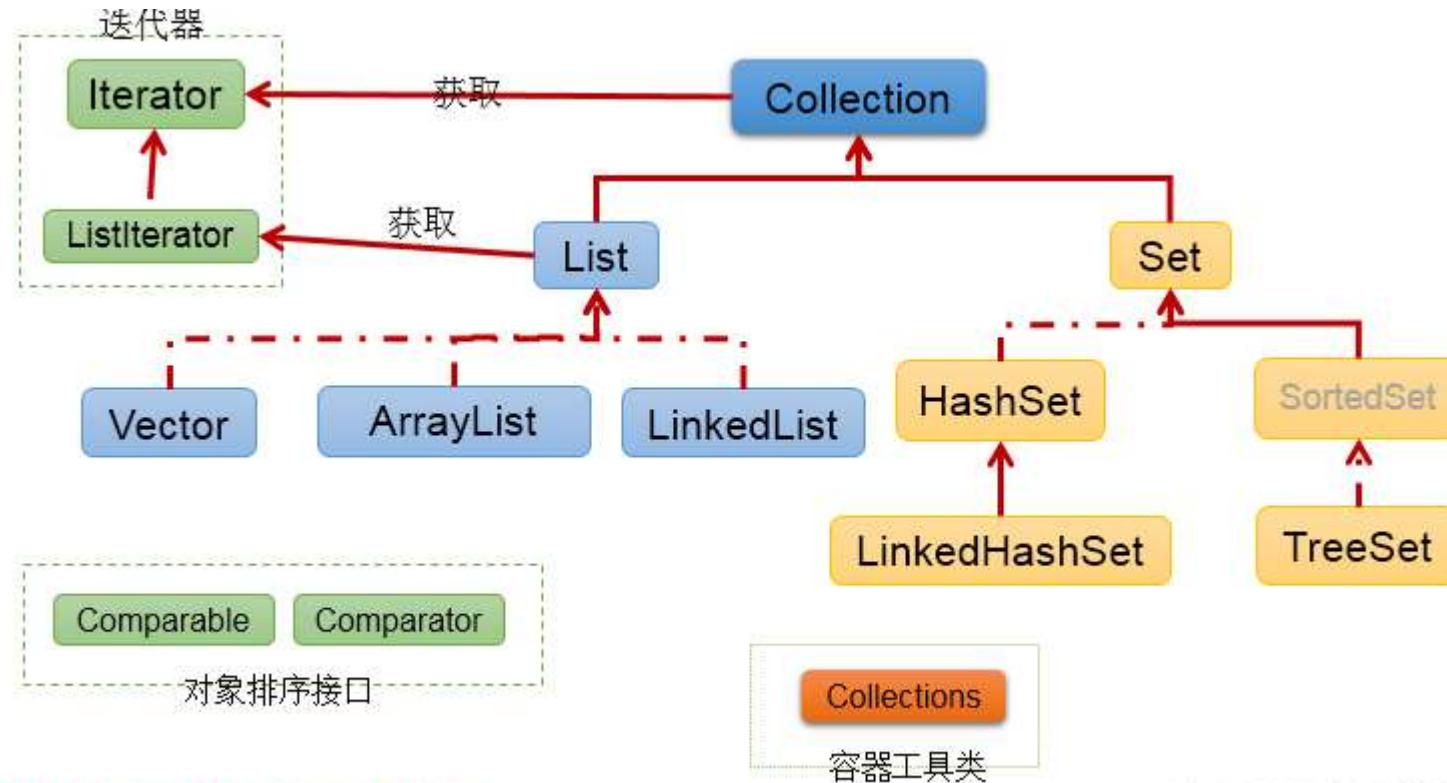
1. 单列集合框架结构

```

----Collection接口: 单列集合, 用来存储一个一个的对象
*      |----List接口: 存储序的、可重复的数据。-->“动态”数组
*          |----ArrayList、LinkedList、Vector
*
*      |----Set接口: 存储无序的、不可重复的数据 -->高中讲的“集合”
*          |----HashSet、LinkedHashSet、TreeSet

```

对应图示：



JDK提供的集合API位于java.util包内

2. Collection接口常用方法:

```

add(Object obj),addAll(Collection coll),size(),isEmpty(),clear();
contains(Object obj),containsAll(Collection coll),remove(Object
obj),removeAll(Collection coll),retainsAll(Collection coll),equals(Object
obj);

hashCode(),toArray(),iterator();

```

3. Collection集合与数组间的转换

```

//集合 --->数组: toArray()
Object[] arr = coll.toArray();
for(int i = 0;i < arr.length;i++){
    System.out.println(arr[i]);
}

```

```

//拓展: 数组 --->集合: 调用Arrays类的静态方法asList(T ... t)
List<String> list = Arrays.asList(new String[]{"AA", "BB", "CC"});
System.out.println(list);

```

```

List arr1 = Arrays.asList(new int[]{123, 456});
System.out.println(arr1.size());//1

```

```

List arr2 = Arrays.asList(new Integer[]{123, 456});
System.out.println(arr2.size());//2

```

4. 使用**Collection**集合存储对象，要求对象所属的类满足：

向**Collection**接口的实现类的对象中添加数据**obj**时，要求**obj**所在类要重写**equals()**.

5. 本章节对大家的要求：

层次一：选择合适的集合类去实现数据的保存，调用其内部的相关方法。

层次二：不同的集合类底层的数据结构为何？如何实现数据的操作的：增删改查等。

Iterator接口与foreach循环

1. 遍历Collection的两种方式：

① 使用迭代器Iterator ② foreach循环（或增强for循环）

2. java.util包下定义的迭代器接口： Iterator

2.1 说明：

- Iterator对象称为迭代器(设计模式的一种)，主要用于遍历 Collection 集合中的元素。
- GOF给迭代器模式的定义为：提供一种方法访问一个容器(container)对象中各个元素，而又不需暴露该对象的内部细节。迭代器模式，就是为容器而生。

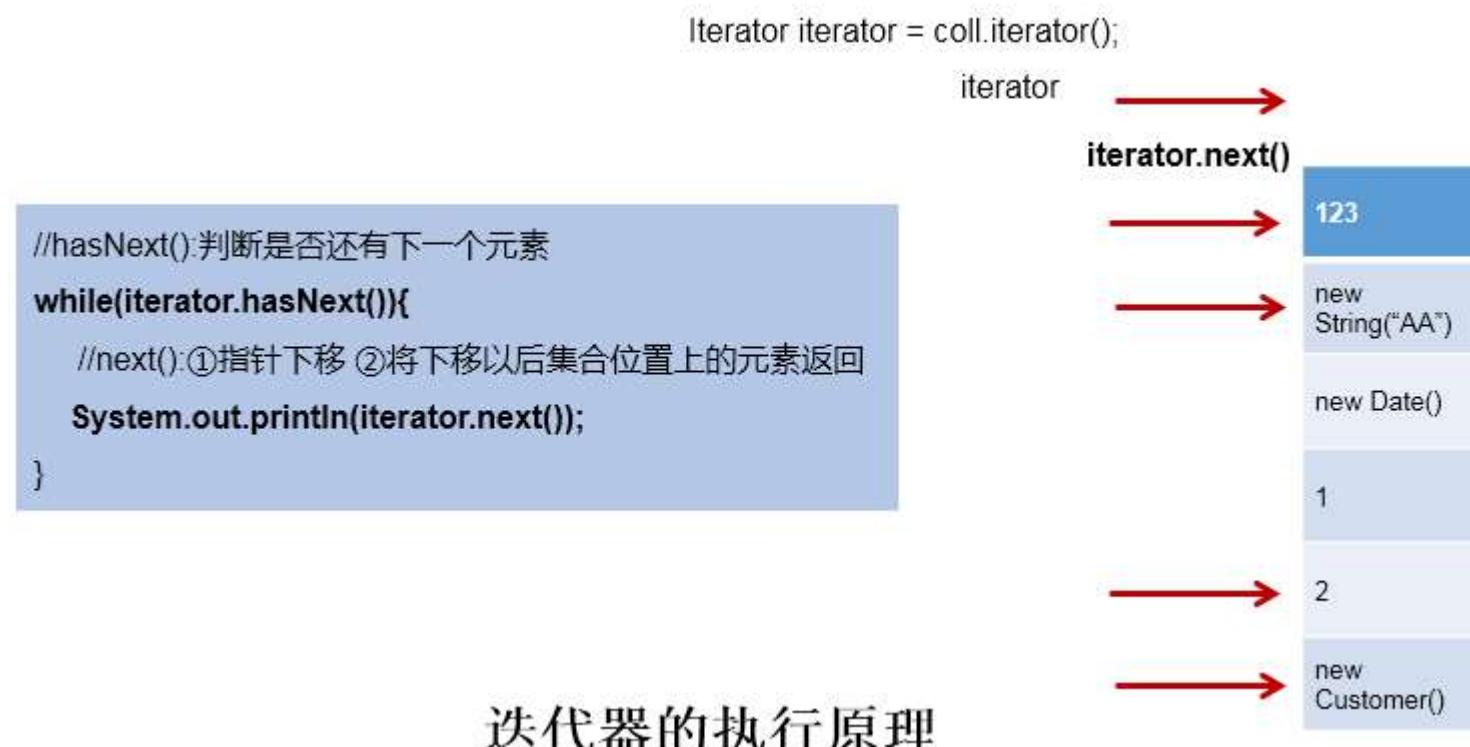
2.2 作用：遍历集合Collectiton元素

2.3 如何获取实例： coll.iterator()返回一个迭代器实例

2.4 遍历的代码实现：

```
Iterator iterator = coll.iterator();
//hasNext(): 判断是否还有下一个元素
while(iterator.hasNext()){
    //next(): ①指针下移 ②将下移以后集合位置上的元素返回
    System.out.println(iterator.next());
}
```

2.5 图示说明：



2.6 remove()的使用：

```
// 测试Iterator中的remove()
// 如果还未调用next()或在上一次调用 next 方法之后已经调用了 remove 方法，再调用 remove 都会报IllegalStateException。
// 内部定义了remove()，可以在遍历的时候，删除集合中的元素。此方法不同于集合直接调用remove()
@Test
public void test3(){
    Collection coll = new ArrayList();
    coll.add(123);
    coll.add(456);
    coll.add(new Person("Jerry", 20));
    coll.add(new String("Tom"));
    coll.add(false);

    // 删除集合中"Tom"
}
```

```

Iterator iterator = coll.iterator();
while (iterator.hasNext()){
    iterator.remove();
    Object obj = iterator.next();
    if("Tom".equals(obj)){
        iterator.remove();
    }
}

//遍历集合
iterator = coll.iterator();
while (iterator.hasNext()){
    System.out.println(iterator.next());
}
}

```

3.jdk5.0新特性--增强for循环: (foreach循环)

1. 遍历集合举例:

```

@Test
public void test1(){
    Collection coll = new ArrayList();
    coll.add(123);
    coll.add(456);
    coll.add(new Person("Jerry", 20));
    coll.add(new String("Tom"));
    coll.add(false);

    //for(集合元素的类型 局部变量 : 集合对象)

    for(Object obj : coll){
        System.out.println(obj);
    }
}

```

说明:
内部仍然调用了迭代器。

2. 遍历数组举例:

```

@Test
public void test2(){
    int[] arr = new int[]{1,2,3,4,5,6};
    //for(数组元素的类型 局部变量 : 数组对象)
    for(int i : arr){
        System.out.println(i);
    }
}

```

1. 存储的数据特点：存储序的、可重复的数据。

2. 常用方法：(记住)

增：`add(Object obj)`
 删：`remove(int index) / remove(Object obj)`
 改：`set(int index, Object ele)`
 查：`get(int index)`
 插：`add(int index, Object ele)`
 长度：`size()`
 遍历：
 ① `Iterator`迭代器方式
 ② 增强`for`循环
 ③ 普通的循环

3. 常用实现类：

```
|----Collection接口：单列集合，用来存储一个一个的对象
* |----List接口：存储序的、可重复的数据。-->“动态”数组，替换原的数组
*   |----ArrayList：作为List接口的主要实现类；线程不安全的，效率高；底层使用
Object[] elementData存储
*   |----LinkedList：对于频繁的插入、删除操作，使用此类效率比ArrayList高；底层使
用双向链表存储
*   |----Vector：作为List接口的古老实现类；线程安全的，效率低；底层使用Object[]
elementData存储
```

4. 源码分析(难点)

4.1 ArrayList的源码分析：

```
* 2.1 jdk 7情况下
*   ArrayList list = new ArrayList(); //底层创建了长度是10的Object[]数组
elementData
*   list.add(123); //elementData[0] = new Integer(123);
*
* ...
*   list.add(11); //如果此次的添加导致底层elementData数组容量不够，则扩容。
*   默认情况下，扩容为原来的容量的1.5倍，同时需要将原有数组中的数据复制到新的
数组中。
*
*   结论：建议开发中使用带参的构造器：ArrayList list = new ArrayList(int
capacity)
*
```

2.2 jdk 8中ArrayList的变化：

```
*   ArrayList list = new ArrayList(); //底层Object[] elementData初始化为
{}。并没创建长度为10的数组
*
*   list.add(123); //第一次调用add()时，底层才创建了长度10的数组，并将数据
123添加到elementData[0]
*
* ...
*   后续的添加和扩容操作与jdk 7 无异。
```

* 2.3 小结：jdk7中的ArrayList的对象的创建类似于单例的饿汉式，而jdk8中的ArrayList的对象

* 的创建类似于单例的懒汉式，延迟了数组的创建，节省内存。

4.2 LinkedList的源码分析：

```
*   LinkedList list = new LinkedList(); 内部声明了Node类型的first和last
属性，默认值为null
*   list.add(123); //将123封装到Node中，创建了Node对象。
*
```

```
*      其中，Node定义为：体现了LinkedList的双向链表的说法
*      private static class Node<E> {
    E item;
    Node<E> next;
    Node<E> prev;

    Node(Node<E> prev, E element, Node<E> next) {
        this.item = element;
        this.next = next;
        this.prev = prev;
    }
}
```

4.3 Vector的源码分析：

jdk7和jdk8中通过Vector()构造器创建对象时，底层都创建了长度为10的数组。在扩容方面，默认扩容为原来的数组长度的2倍。

5. 存储的元素的要求：

添加的对象，所在的类要重写equals()方法

[面试题]

- * 面试题：ArrayList、LinkedList、Vector者的异同？
- * 同：三个类都是实现了List接口，存储数据的特点相同：存储序的、可重复的数据
- * 不同：见上（第3部分+第4部分）

1. 存储的数据特点：无序的、不可重复的元素

具体的：

以`HashSet`为例说明：

1. 无序性：不等于随机性。存储的数据在底层数组中并非照数组索引的顺序添加，而是根据数据的哈希值决定的。
2. 不可重复性：保证添加的元素照`equals()`判断时，不能返回`true`. 即：相同的元素只能添加一个。

2. 元素添加过程：(以`HashSet`为例)

我们向`HashSet`中添加元素`a`, 首先调用元素`a`所在类的`hashCode()`方法，计算元素`a`的哈希值，

此哈希值接着通过某种算法计算出在`HashSet`底层数组中的存放位置（即为：索引位置，判断数组此位置上是否已经元素：

如果此位置上没其他元素，则元素`a`添加成功。--->情况1

如果此位置上其他元素`b`(或以链表形式存在的多个元素，则比较元素`a`与元素`b`的`hash`值：

如果`hash`值不相同，则元素`a`添加成功。--->情况2

如果`hash`值相同，进而需要调用元素`a`所在类的`equals()`方法：

`equals()`返回`true`, 元素`a`添加失败

`equals()`返回`false`, 则元素`a`添加成功。--->情况2

对于添加成功的情况2和情况3而言：元素`a` 与已经存在指定索引位置上数据以链表的方式存储。

`jdk 7` : 元素`a`放到数组中，指向原来的元素。

`jdk 8` : 原来的元素在数组中，指向元素`a`

总结：七上八下

`HashSet`底层：数组+链表的结构。（前提：`jdk7`）

3. 常用方法

`Set`接口中没额外定义新的方法，使用的都是`Collection`中声明过的方法。

4. 常用实现类：

```
|----Collection接口：单列集合，用来存储一个一个的对象
*      |----Set接口：存储无序的、不可重复的数据 --->高中讲的“集合”
*          |----HashSet：作为Set接口的主要实现类；线程不安全的；可以存储null值
*              |----LinkedHashSet：作为HashSet的子类；遍历其内部数据时，可以按照添加的顺序遍历
*                  在添加数据的同时，每个数据还维护了两个引用，记录此数据前一个数据和后一个数据。
*                      对于频繁的遍历操作，LinkedHashSet效率高于
* HashSet.
*          |----TreeSet：可以按照添加对象的指定属性，进行排序。
```

5. 存储对象所在类的要求：

`HashSet/LinkedHashSet`:

要求：向`Set`（主要指：`HashSet`、`LinkedHashSet`）中添加的数据，其所在的类一定要重写`hashCode()`和`equals()`

要求：重写的`hashCode()`和`equals()`尽可能保持一致性：相等的对象必须具有相等的散列码

* 重写两个方法的小技巧：对象中用作`equals()`方法比较的`Field`，都应该用来计算`hashCode` 值。

*

TreeSet:

1. 自然排序中，比较两个对象是否相同的标准为：`compareTo()`返回0. 不再是`equals()`.
2. 定制排序中，比较两个对象是否相同的标准为：`compare()`返回0. 不再是`equals()`.

6. TreeSet的使用**6.1 使用说明：**

1. 向`TreeSet`中添加的数据，要求是相同类的对象。
2. 两种排序方式：自然排序（实现`Comparable`接口）和 定制排序（`Comparator`）

6.2 常用的排序方式：

//方式一：自然排序

`@Test`

```
public void test1(){
    TreeSet set = new TreeSet();
```

//失败：不能添加不同类的对象

```
//    set.add(123);
//    set.add(456);
//    set.add("AA");
//    set.add(new User("Tom",12));
```

//举例一：

```
//    set.add(34);
//    set.add(-34);
//    set.add(43);
//    set.add(11);
//    set.add(8);
```

//举例二：

```
set.add(new User("Tom",12));
set.add(new User("Jerry",32));
set.add(new User("Jim",2));
set.add(new User("Mike",65));
set.add(new User("Jack",33));
set.add(new User("Jack",56));
```

```
Iterator iterator = set.iterator();
```

```
while(iterator.hasNext()){
    System.out.println(iterator.next());
```

```
}
```

```
}
```

//方式二：定制排序

`@Test``public void test2(){`

```
    Comparator com = new Comparator() {
```

//照年龄从小到大排列

`@Override`

```
    public int compare(Object o1, Object o2) {
```

```
        if(o1 instanceof User && o2 instanceof User){
```

```
            User u1 = (User)o1;
```

```
            User u2 = (User)o2;
```

```
            return Integer.compare(u1.getAge(),u2.getAge());
```

`}else{`

```
            throw new RuntimeException("输入的数据类型不匹配");
```

```
        }
```

```
}
```

```
};
```

```
TreeSet set = new TreeSet(com);
set.add(new User("Tom",12));
set.add(new User("Jerry",32));
set.add(new User("Jim",2));
set.add(new User("Mike",65));
set.add(new User("Mary",33));
set.add(new User("Jack",33));
set.add(new User("Jack",56));

Iterator iterator = set.iterator();
while(iterator.hasNext()){
    System.out.println(iterator.next());
}
}
```

双列集合框架: Map

1. 常用实现类结构

|----Map: 双列数据, 存储key-value对的数据 ---类似于高中的函数: $y = f(x)$
 * |----HashMap: 作为Map的主要实现类; 线程不安全的, 效率高; 存储null的key和
 value
 * |----LinkedHashMap: 保证在遍历map元素时, 可以照添加的顺序实现遍
 历。
 * 原因: 在原的HashMap底层结构基础上, 添加了一对指针, 指向前
 一个和后一个元素。
 * 对于频繁的遍历操作, 此类执行效率高于HashMap。
 * |----TreeMap: 保证照添加的key-value对进行排序, 实现排序遍历。此时考虑key
 的自然排序或定制排序
 * 底层使用红黑树
 * |----Hashtable: 作为古老的实现类; 线程安全的, 效率低; 不能存储null的key
 和value
 * |----Properties: 常用来处理配置文件。key和value都是String类型
 *
 *
 * HashMap的底层: 数组+链表 (jdk7及之前)
 * 数组+链表+红黑树 (jdk 8)

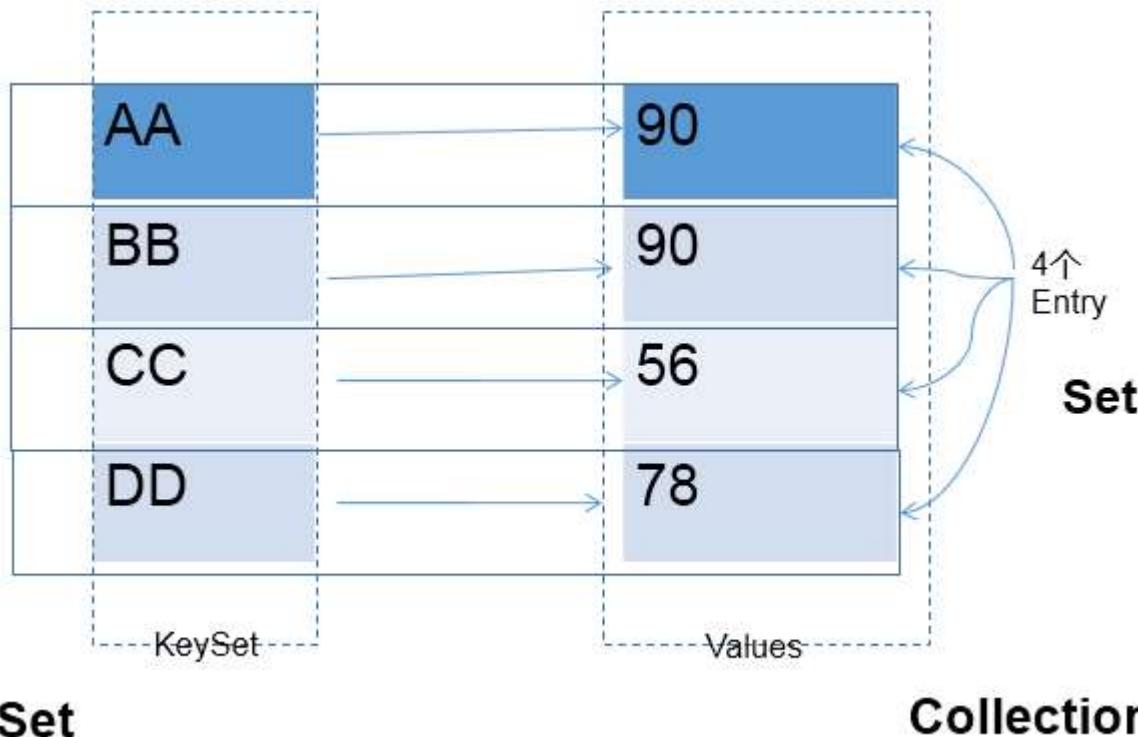
[面试题]

- * 1. HashMap的底层实现原理?
- * 2. HashMap 和 Hashtable的异同?
- * 3. CurrentHashMap 与 Hashtable的异同? (暂时不讲)

2. 存储结构的理解:

>Map 中的key: 无序的、不可重复的, 使用Set存储所的key ---> key所在的类要重写
`equals()`和`hashCode()` (以HashMap为例)
 >Map 中的value: 无序的、可重复的, 使用Collection存储所的value ---> value所在的类要
 重写`equals()`
 > 一个键值对: key-value构成了一个Entry对象。
 >Map 中的entry: 无序的、不可重复的, 使用Set存储所的entry

图示:



3. 常用方法

- * 添加: `put(Object key, Object value)`

```
* 删除: remove(Object key)
* 修改: put(Object key, Object value)
* 查询: get(Object key)
* 长度: size()
* 遍历: keySet() / values() / entrySet()
```

4. 内存结构说明: (难点)

4.1 HashMap在jdk7中实现原理:

```
HashMap map = new HashMap():
* 在实例化以后, 底层创建了长度是16的一维数组Entry[] table。
* ... 可能已经执行过多次put...
* map.put(key1,value1):
* 首先, 调用key1所在类的hashCode()计算key1哈希值, 此哈希值经过某种算法计算
以后, 得到在Entry数组中的存放位置。
* 如果此位置上的数据为空, 此时的key1-value1添加成功。 ---- 情况1
* 如果此位置上的数据不为空, (意味着此位置上存在一个或多个数据(以链表形式存
在)), 比较key1和已经存在的一个或多个数据的哈希值:
* 如果key1的哈希值与已经存在的数据的哈希值都不相同, 此时key1-
value1添加成功。 ---- 情况2
* 如果key1的哈希值和已经存在的某一个数据(key2-value2)的哈希值相
同, 继续比较: 调用key1所在类的equals(key2)方法, 比较:
* 如果equals()返回false: 此时key1-value1添加成功。 ---- 情况3
* 如果equals()返回true: 使用value1替换value2。
*
* 补充: 关于情况2和情况3: 此时key1-value1和原来的数据以链表的方式存储。
*
* 在不断的添加过程中, 会涉及到扩容问题, 当超出临界值(且要存放的位置非空)
时, 扩容。默认的扩容方式: 扩容为原来容量的2倍, 并将原的数据复制过来。
```

4.2 HashMap在jdk8中相较于jdk7在底层实现方面的不同:

1. new HashMap(): 底层没创建一个长度为16的数组
 2. jdk 8底层的数组是: Node[], 而非Entry[]
 3. 首次调用put()方法时, 底层创建长度为16的数组
 4. jdk7底层结构只: 数组+链表。 jdk8中底层结构: 数组+链表+红黑树。
- 4.1 形成链表时, 七上八下 (jdk7: 新的元素指向旧的元素。 jdk8: 旧的元素指向新的元素)
- 4.2 当数组的某一个索引位置上的元素以链表形式存在的数据个数 > 8 且当前数组的长度 > 64时, 此时此索引位置上的所数据改为使用红黑树存储。

4.3 HashMap底层典型属性的属性的说明:

`DEFAULT_INITIAL_CAPACITY` : HashMap的默认容量, 16
`DEFAULT_LOAD_FACTOR`: HashMap的默认加载因子: 0.75
`threshold`: 扩容的临界值, =容量*填充因子: $16 * 0.75 \Rightarrow 12$
`TREEIFY_THRESHOLD`: Bucket中链表长度大于该默认值, 转化为红黑树: 8
`MIN_TREEIFY_CAPACITY`: 桶中的Node被树化时最小的hash表容量: 64

4.4 LinkedHashMap的底层实现原理(了解)

LinkedHashMap底层使用的结构与HashMap相同, 因为LinkedHashMap继承于HashMap. 区别就在于: LinkedHashMap内部提供了Entry, 替换HashMap中的Node.

HashMap中的内部类：Node

```
static class Node<K,V> implements Map.Entry<K,V> {
    final int hash;
    final K key;
    V value;
    Node<K,V> next;
}
```

LinkedHashMap中的内部类：Entry

```
static class Entry<K,V> extends HashMap.Node<K,V> {
    Entry<K,V> before, after;
    Entry(int hash, K key, V value, Node<K,V> next) {
        super(hash, key, value, next);
    }
}
```

5. TreeMap的使用

//向TreeMap中添加key-value，要求key必须是由同一个类创建的对象
//因为要照key进行排序：自然排序、定制排序

6. 使用Properties读取配置文件

```
//Properties: 常用来处理配置文件。key和value都是String类型
public static void main(String[] args) {
    FileInputStream fis = null;
    try {
        Properties pros = new Properties();

        fis = new FileInputStream("jdbc.properties");
        pros.load(fis); //加载流对应的文件

        String name = pros.getProperty("name");
        String password = pros.getProperty("password");

        System.out.println("name = " + name + ", password = " +
password);
    } catch (IOException e) {
        e.printStackTrace();
    } finally {
        if(fis != null){
            try {
                fis.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Collections工具类的使用

Collections工具类**1. 作用：操作Collection和Map的工具类****2. 常用方法：**

- ① **reverse(List):** 反转 List 中元素的顺序
- ① **shuffle(List):** 对 List 集合元素进行随机排序
- ① **sort(List):** 根据元素的自然顺序对指定 List 集合元素升序排序
- ① **sort(List, Comparator):** 根据指定的 Comparator 产生的顺序对 List 集合元素进行排序
- ① **swap(List, int, int):** 将指定 list 集合中的 i 处元素和 j 处元素进行交换
- ① **Object max(Collection):** 根据元素的自然顺序，返回给定集合中的最大元素
- ① **Object max(Collection, Comparator):** 根据 Comparator 指定的顺序，返回给定集合中的最大元素
- ① **Object min(Collection):**
- ① **Object min(Collection, Comparator):**
- ① **int frequency(Collection, Object):** 返回指定集合中指定元素的出现次数
- ① **void copy(List dest, List src):** 将src中的内容复制到dest中
- ① **boolean replaceAll(List list, Object oldVal, Object newVal):** 使用新值替换 List 对象的所旧值

<code><T> static Collection<T> synchronizedCollection(Collection<T> c)</code>	Returns a synchronized (thread-safe) collection backed by the specified collection.
<code><T> static List<T> synchronizedList(List<T> list)</code>	Returns a synchronized (thread-safe) list backed by the specified list.
<code><K, V> static Map<K, V> synchronizedMap(Map<K, V> m)</code>	Returns a synchronized (thread-safe) map backed by the specified map.
<code><T> static Set<T> synchronizedSet(Set<T> s)</code>	Returns a synchronized (thread-safe) set backed by the specified set.
<code><K, V> static SortedMap<K, V> synchronizedSortedMap(SortedMap<K, V> m)</code>	Returns a synchronized (thread-safe) sorted map backed by the specified sorted map.
<code><T> static SortedSet<T> synchronizedSortedSet(SortedSet<T> s)</code>	Returns a synchronized (thread-safe) sorted set backed by the specified sorted set.

说明：ArrayList和HashMap都是线程不安全的，如果程序要求线程安全，我们可以将 ArrayList、HashMap转换为线程的。
使用synchronizedList(List list) 和 synchronizedMap(Map map)

3. 面试题：

面试题：*Collection 和 Collections的区别？*

1. 数据结构概述

数据结构（Data Structure）是一门和计算机硬件与软件都密切相关的学科，它的研究重点是在计算机的程序设计领域中探讨如何在计算机中组织和存储数据并进行高效率的运用，**涉及的内容包含：数据的逻辑关系、数据的存储结构、排序算法（Algorithm）、查找（或搜索）等。**

2. 数据结构与算法的理解

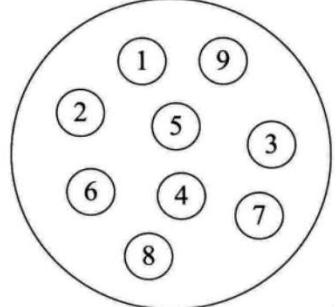
程序能否快速而高效地完成预定的任务，取决于是否选对了数据结构，而程序是否能清楚而正确地把问题解决，则取决于算法。

所以大家认为：“**Algorithms + Data Structures = Programs**”（出自：Pascal之父 Nicklaus Wirth）

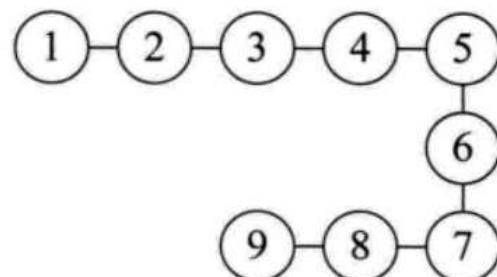
总结：算法是为了解决实际问题而设计的，数据结构是算法需要处理的问题载体。

3. 数据结构的研究对象

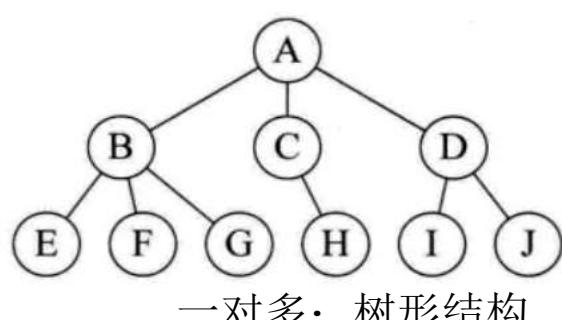
3.1 数据间的逻辑结构



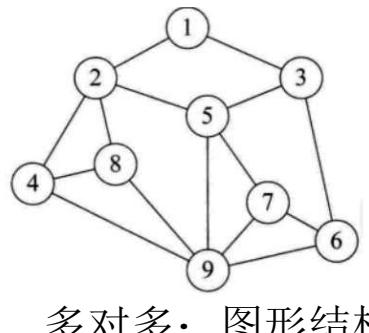
集合结构



一对一：线性结构



一对多：树形结构



多对多：图形结构

3.2 数据的存储结构：

线性表（顺序表、链表、栈、队列）

树
图

说明：习惯上把顺序表和链表看做基本数据结构（或真实数据结构）

习惯上把栈、队列、树、图看做抽象数据类型，简称ADT

4. 使用详情见思维导图：

《附录：尚硅谷_宋红康_数据结构概述-Java版.xmind》

1. 泛型的概念

所谓泛型，就是允许在定义类、接口时通过一个标识表示类中某个属性的类型或者
是某个方法的返
回值及参数类型。这个类型参数将在使用时（例如，继承或实现这个接口，用这个
类型声明变量、
创建对象时确定（即传入实际的类型参数，也称为类型实参）。

2. 泛型的引入背景

集合容器类在设计阶段/声明阶段不能确定这个容器到底实际存的是什么类型的对象，所以
在JDK1.5之前只能把元素类型设计为Object，JDK1.5之后使用泛型来解决。因为这个时候
除了元素的类型不确定，其他的部分是确定的，例如关于这个元素如何保存，如何管理等
是确定的，因此此时把元素的类型设计成一个参数，这个类型参数叫做泛型。

Collection<E>, List<E>, ArrayList<E> 这个<E>就是类型参数，即泛型。

泛型在集合中的使用

1. 在集合中使用泛型之前的例子

```

@Test
public void test1(){
    ArrayList list = new ArrayList();
    //需求：存放学生的成绩
    list.add(78);
    list.add(76);
    list.add(89);
    list.add(88);
    //问题一：类型不安全
    //      list.add("Tom");

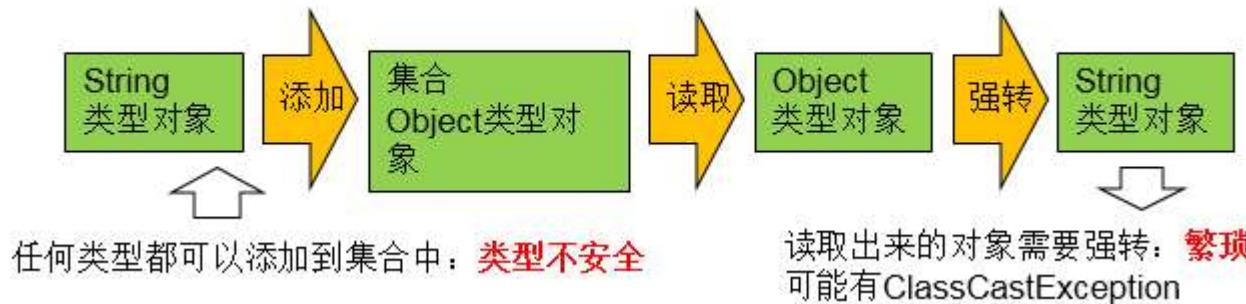
    for(Object score : list){
        //问题二：强转时，可能出现ClassCastException
        int stuScore = (Integer) score;

        System.out.println(stuScore);
    }
}

```

图示：

在集合中没有泛型时



2. 在集合中使用泛型例子1

```

@Test
public void test2(){
    ArrayList<Integer> list = new ArrayList<Integer>();

    list.add(78);
    list.add(87);
    list.add(99);
    list.add(65);
    //编译时，就会进行类型检查，保证数据的安全
    //      list.add("Tom");

    //方式一：
    //      for(Integer score : list){
    //          //避免了强转操作
    //          int stuScore = score;
    //
    //          System.out.println(stuScore);
    //
    //      }

    //方式二：
    Iterator<Integer> iterator = list.iterator();
    while(iterator.hasNext()){
        int stuScore = iterator.next();
        System.out.println(stuScore);
    }
}

```

图示：

在集合中有泛型时



3. 在集合中使用泛型例子2

```
// 在集合中使用泛型的情况：以HashMap为例
@Test
public void test3(){
//      Map<String, Integer> map = new HashMap<String, Integer>();
// jdk7新特性：类型推断
Map<String, Integer> map = new HashMap<>();

map.put("Tom", 87);
map.put("Jerry", 87);
map.put("Jack", 67);

//      map.put(123, "ABC");
// 泛型的嵌套
Set<Map.Entry<String, Integer>> entry = map.entrySet();
Iterator<Map.Entry<String, Integer>> iterator = entry.iterator();

while(iterator.hasNext()){
    Map.Entry<String, Integer> e = iterator.next();
    String key = e.getKey();
    Integer value = e.getValue();
    System.out.println(key + "----" + value);
}
}
```

4. 集合中使用泛型总结：

- * ① 集合接口或集合类在jdk5.0时都修改为带泛型的结构。
- * ② 在实例化集合类时，可以指明具体的泛型类型
- * ③ 指明完以后，在集合类或接口中凡是定义类或接口时，内部结构（比如：方法、构造器、属性等）使用到类的泛型的位置，都指定为实例化的泛型类型。
- * 比如：add(E e) ---> 实例化以后：add(Integer e)
- * ④ 注意点：泛型的类型必须是类，不能是基本数据类型。需要用到基本数据类型的位
置，拿包装类替换
- * ⑤ 如果实例化时，没指明泛型的类型。默认类型为java.Lang.Object类型。

1. 举例：

【Order.java】

```

public class Order<T> {

    String orderName;
    int orderId;

    //类的内部结构就可以使用类的泛型

    T orderT;

    public Order(){
        //编译不通过
//        T[] arr = new T[10];
        //编译通过
        T[] arr = (T[]) new Object[10];
    }

    public Order(String orderName,int orderId,T orderT){
        this.orderName = orderName;
        this.orderId = orderId;
        this.orderT = orderT;
    }

    //如下的个方法都不是泛型方法
    public T getOrderT(){
        return orderT;
    }

    public void setOrderT(T orderT){
        this.orderT = orderT;
    }

    @Override
    public String toString() {
        return "Order{" +
            "orderName='" + orderName + '\'' +
            ", orderId=" + orderId +
            ", orderT=" + orderT +
            '}';
    }
    //静态方法中不能使用类的泛型。
//    public static void show(T orderT){
//        System.out.println(orderT);
//    }

    public void show(){
        //编译不通过
//        try{
//        //
//        //
//        }catch(T t){
//        //
//        }
    }

    //泛型方法：在方法中出现了泛型的结构，泛型参数与类的泛型参数没关系。
    //换句话说，泛型方法所属的类是不是泛型类都没关系。
}

```

```
//泛型方法，可以声明为静态的。原因：泛型参数是在调用方法时确定的。并非在实例化类时确定。
public static <E> List<E> copyFromArrayList(E[] arr){

    ArrayList<E> list = new ArrayList<>();

    for(E e : arr){
        list.add(e);
    }
    return list;

}
```

【SubOrder.java】

```
public class SubOrder extends Order<Integer> {//SubOrder：不是泛型类
```

```
public static <E> List<E> copyFromArrayList(E[] arr){

    ArrayList<E> list = new ArrayList<>();

    for(E e : arr){
        list.add(e);
    }
    return list;

}
```

//实例化时，如下的代码是错误的

```
SubOrder<Integer> o = new SubOrder<>();
```

【SubOrder1.java】

```
public class SubOrder1<T> extends Order<T> {//SubOrder1<T>：仍然是泛型类

}
```

【测试】

```
@Test
```

```
public void test1(){
    //如果定义了泛型类，实例化没指明类的泛型，则认为此泛型类型为Object类型
    //要求：如果大家定义了类是带泛型的，建议在实例化时要指明类的泛型。
    Order order = new Order();
    order.setOrderT(123);
    order.setOrderT("ABC");

    //建议：实例化时指明类的泛型
    Order<String> order1 = new Order<String>
    ("orderAA",1001,"order:AA");

    order1.setOrderT("AA:hello");
}
```

```
@Test
```

```
public void test2(){
    SubOrder sub1 = new SubOrder();
```

```

//由于子类在继承带泛型的父类时，指明了泛型类型。则实例化子类对象时，不再
需要指明泛型。
sub1.setOrderT(1122);

SubOrder1<String> sub2 = new SubOrder1<>();
sub2.setOrderT("order2...");

}

@Test
public void test3(){

    ArrayList<String> list1 = null;
    ArrayList<Integer> list2 = new ArrayList<Integer>();
    //泛型不同的引用不能相互赋值。
    //    list1 = list2;

    Person p1 = null;
    Person p2 = null;
    p1 = p2;

}

//测试泛型方法
@Test
public void test4(){
    Order<String> order = new Order<>();
    Integer[] arr = new Integer[]{1,2,3,4};
    //泛型方法在调用时，指明泛型参数的类型。
    List<Integer> list = order.copyFromArrayToList(arr);

    System.out.println(list);
}

```

2. 注意点：

1. 泛型类可能有多个参数，此时应将多个参数一起放在尖括号内。比如：
`<E1,E2,E3>`
2. 泛型类的构造器如下： `public GenericClass(){}。`
而下面是错误的：`public GenericClass<E>(){}`
3. 实例化后，操作原来泛型位置的结构必须与指定的泛型类型一致。
4. 泛型不同的引用不能相互赋值。
>尽管在编译时 `ArrayList<String>` 和 `ArrayList<Integer>` 是两种类型，但是，在运行时只有一个 `ArrayList` 被加载到 JVM 中。
5. 泛型如果不指定，将被擦除，泛型对应的类型均按照 `Object` 处理，但不等价于 `Object`。 **经验：** 泛型要使用一路都用。要不用，一路都不要用。
6. 如果泛型结构是一个接口或抽象类，则不可创建泛型类的对象。
7. jdk1.7，泛型的简化操作： `ArrayList<Fruit> fList = new ArrayList<>();`
8. 泛型的指定中不能使用基本数据类型，可以使用包装类替换。

9. 在类/接口上声明的泛型，在本类或本接口中即代表某种类型，可以作为非静态属性的类型、非静态方法的参数类型、非静态方法的返回值类型。但在**静态方法中不能使用类的泛型。**

10. 异常类不能是泛型的

11. 不能使用new E[]。但是可以：E[] elements = (E[])new Object[capacity];

参考：ArrayList源码中声明：Object[] elementData，而非泛型参数类型数组。

12. 父类有泛型，子类可以选择保留泛型也可以选择指定泛型类型：

- 子类不保留父类的泛型：按需实现

- 没有类型 擦除

- 具体类型

- 子类保留父类的泛型：泛型子类

- 全部保留

- 部分保留

结论：子类必须是“富二代”，子类除了指定或保留父类的泛型，还可以增加自己的泛型

2.3.2 泛型的实现

3. 应用场景举例：

【DAO.java】：定义了操作数据库中的表的通用操作。 ORM思想(数据库中的表和Java中的类对应)

```
public class DAO<T> { // 表的共性操作的DAO
```

```
// 添加一条记录
public void add(T t){
}

// 删除一条记录
public boolean remove(int index){
    return false;
}

// 修改一条记录
public void update(int index, T t){
}

// 查询一条记录
public T getIndex(int index){
    return null;
}

// 查询多条记录
public List<T> getForList(int index){
    return null;
}

// 泛型方法
// 举例：获取表中一共有多少条记录？获取最大的员工入职时间？
public <E> E getValue(){
    return null;
}
```

【CustomerDAO.java】：

```
public class CustomerDAO extends DAO<Customer>{//只能操作某一个表的DAO
}
```

【StudentDAO.java】：

```
public class StudentDAO extends DAO<Student> {//只能操作某一个表的DAO
}
```

泛型在继承上的体现

泛型在继承上的体现：

```
/*
 1. 泛型在继承方面的体现
```

虽然类A是类B的父类，但是G<A> 和G二者不具备父子类关系，二者是并列关系。

补充：类A是类B的父类，A<G> 是 B<G> 的父类

```
/*
@Test
public void test1(){

    Object obj = null;
    String str = null;
    obj = str;

    Object[] arr1 = null;
    String[] arr2 = null;
    arr1 = arr2;
    //编译不通过
    // Date date = new Date();
    // str = date;
    List<Object> list1 = null;
    List<String> list2 = new ArrayList<String>();
    //此时的list1和list2的类型不具父子类关系
    //编译不通过
    // list1 = list2;
    /*
    反证法：
    假设list1 = list2;
    list1.add(123); 导致混入非String的数据。出错。
    */

    show(list1);
    show1(list2);

}
```

```
public void show1(List<String> list){
```

```
}
```

```
public void show(List<Object> list){
```

```
}
```

```
@Test
```

```
public void test2(){
```

```
AbstractList<String> list1 = null;
List<String> list2 = null;
ArrayList<String> list3 = null;
```

```
list1 = list3;
list2 = list3;
```

```
List<String> list4 = new ArrayList<>();
```

}

1. 通配符的使用

```

/*
    通配符的使用
    通配符: ?

    类A是类B的父类, G<A>和G<B>是没关系的, 二者共同的父类是: G<?>

*/
@Test
public void test3(){
    List<Object> list1 = null;
    List<String> list2 = null;

    List<?> list = null;

    list = list1;
    list = list2;
    //编译通过
    //    print(list1);
    //    print(list2);

    //
    List<String> list3 = new ArrayList<>();
    list3.add("AA");
    list3.add("BB");
    list3.add("CC");
    list = list3;
    //添加(写入): 对于List<?>就不能向其内部添加数据。
    //除了添加null之外。
    //    list.add("DD");
    //    list.add('?');

    list.add(null);

    //获取(读取): 允许读取数据, 读取的数据类型为Object。
    Object o = list.get(0);
    System.out.println(o);

}

public void print(List<?> list){
    Iterator<?> iterator = list.iterator();
    while(iterator.hasNext()){
        Object obj = iterator.next();
        System.out.println(obj);
    }
}

```

2. 涉及通配符的集合的数据的写入和读取:

见上

3. 有限制条件的通配符的使用

```

/*
    限制条件的通配符的使用。

```

? extends A:
G<? extends A> 可以作为G<A>和G的父类，其中B是A的子类

? super A:
G<? super A> 可以作为G<A>和G的父类，其中B是A的父类

```
/*
@Test
public void test4(){

    List<? extends Person> list1 = null;
    List<? super Person> list2 = null;

    List<Student> list3 = new ArrayList<Student>();
    List<Person> list4 = new ArrayList<Person>();
    List<Object> list5 = new ArrayList<Object>();

    list1 = list3;
    list1 = list4;
//    list1 = list5;

//    list2 = list3;
    list2 = list4;
    list2 = list5;

    //读取数据:
    list1 = list3;
    Person p = list1.get(0);
    //编译不通过
    //Student s = list1.get(0);

    list2 = list4;
    Object obj = list2.get(0);
    //编译不通过
//    Person obj = list2.get(0);

    //写入数据:
    //编译不通过
//    list1.add(new Student());

    //编译通过
    list2.add(new Person());
    list2.add(new Student());
}

}
```

File类的使用

1. File类的理解

- * 1. File类的一个对象，代表一个文件或一个文件目录(俗称：文件夹)
- * 2. File类声明在java.io包下
- * 3. File类中涉及到关于文件或文件目录的创建、删除、重命名、修改时间、文件大小等方法，
- * 并未涉及到写入或读取文件内容的操作。如果需要读取或写入文件内容，必须使用IO流来完成。
- * 4. 后续File类的对象常会作为参数传递到流的构造器中，指明读取或写入的“终点”。

2. File的实例化

2.1 常用构造器

```
File(String filePath)
File(String parentPath, String childPath)
File(File parentFile, String childPath)
```

2.2 路径的分类

相对路径：相较于某个路径下，指明的路径。

绝对路径：包含盘符在内的文件或文件目录的路径

说明：

IDEA中：

如果大家开发使用JUnit中的单元测试方法测试，相对路径即为当前Module下。

如果大家使用main()测试，相对路径即为当前的Project下。

Eclipse中：

不管使用单元测试方法还是使用main()测试，相对路径都是当前的Project下。

2.3 路径分隔符

windows和DOS系统默认使用“\”来表示

UNIX和URL使用“/”来表示

3. File类的常用方法

● File类的获取功能

- `public String getAbsolutePath()`: 获取绝对路径
- `public String getPath()`: 获取路径
- `public String getName()`: 获取名称
- `public String getParent()`: 获取上层文件目录路径。若无，返回null
- `public long length()`: 获取文件长度（即：字节数）。不能获取目录的长度。
- `public long lastModified()`: 获取最后一次的修改时间，毫秒值

- `public String[] list()`: 获取指定目录下的所有文件或者文件目录的名称数组
- `public File[] listFiles()`: 获取指定目录下的所有文件或者文件目录的File数组

● File类的重命名功能

- `public boolean renameTo(File dest)`: 把文件重命名为指定的文件路径

● File类的判断功能

- `public boolean isDirectory()`: 判断是否是文件目录
- `public boolean isFile()`: 判断是否是文件
- `public boolean exists()`: 判断是否存在
- `public boolean canRead()`: 判断是否可读
- `public boolean canWrite()`: 判断是否可写
- `public boolean isHidden()`: 判断是否隐藏

● File类的创建功能

- `public boolean createNewFile()`： 创建文件。若文件存在，则不创建，返回`false`
- `public boolean mkdir()`： 创建文件目录。如果此文件目录存在，就不创建了。如果此文件目录的上层目录不存在，也不创建。
- `public boolean mkdirs()`： 创建文件目录。如果上层文件目录不存在，一并创建

注意事项：如果你创建文件或者文件目录没有写盘符路径，那么，默认在项目路径下。

● File类的删除功能

- `public boolean delete()`： 删除文件或者文件夹

删除注意事项：

Java中的删除不走**回收站**。

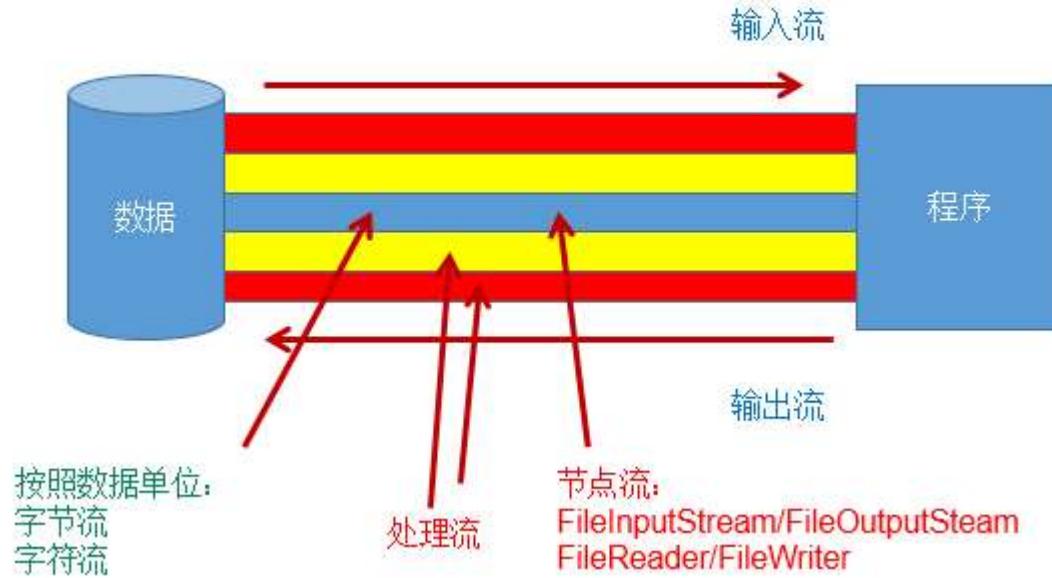
要删除一个文件目录，请注意该文件目录内不能包含文件或者文件目录

1. 流的分类

- * 1. 操作数据单位: 字节流、字符流
- * 2. 数据的流向: 输入流、输出流
- * 3. 流的角色: 节点流、处理流

图示:

流的分类



2. 流的体系结构

分类	字节输入流	字节输出流	字符输入流	字符输出流
抽象基类	<code>InputStream</code>	<code>OutputStream</code>	<code>Reader</code>	<code>Writer</code>
访问文件	<code>FileInputStream</code>	<code>FileOutputStream</code>	<code>FileReader</code>	<code>FileWriter</code>
访问数组	<code>ByteArrayInputStream</code>	<code>ByteArrayOutputStream</code>	<code>CharArrayReader</code>	<code>CharArrayWriter</code>
访问管道	<code>PipedInputStream</code>	<code>PipedOutputStream</code>	<code>PipedReader</code>	<code>PipedWriter</code>
访问字符串			<code>StringReader</code>	<code>StringWriter</code>
缓冲流	<code>BufferedInputStream</code>	<code>BufferedOutputStream</code>	<code>BufferedReader</code>	<code>BufferedWriter</code>
转换流			<code>InputStreamReader</code>	<code>OutputStreamWriter</code>
对象流	<code>ObjectInputStream</code>	<code>ObjectOutputStream</code>		
	<code>FilterInputStream</code>	<code>FilterOutputStream</code>	<code>FilterReader</code>	<code>FilterWriter</code>
打印流		<code>PrintStream</code>		<code>PrinterWriter</code>
推回输入流	<code>PushbackInputStream</code>		<code>PushbackReader</code>	
特殊流	<code>DataInputStream</code>	<code>DataOutputStream</code>		

说明: 红框对应的是IO流中的4个抽象基类。

蓝框的流需要大家重点关注。

3. 重点说明的几个流结构

抽象基类	节点流 (或文件流)	缓冲流 (处理流的一种)
<code>InputStream</code>	<code>FileInputStream (read(byte[] buffer))</code>	<code>BufferedInputStream (read(byte[] buffer))</code>
<code>OutputStream</code>	<code>FileOutputStream (write(byte[] buffer, 0, len))</code>	<code>BufferedOutputStream (write(byte[] buffer, 0, len))</code>
<code>Reader</code>	<code>FileReader (read(char[] cbuf))</code>	<code>BufferedReader (read(char[] cbuf) / readLine())</code>
<code>Writer</code>	<code>FileWriter (write(char[] cbuf, 0, len))</code>	<code>BufferedWriter (write(char[] cbuf, 0, len))</code>

4. 输入、输出的标准化过程

4.1 输入过程

- ① 创建**File**类的对象，指明读取的数据的来源。（要求此文件一定要存在）
- ② 创建相应的输入流，将**File**类的对象作为参数，传入流的构造器中
- ③ 具体的读入过程：

 创建相应的**byte[]** 或 **char[]**。

- ④ 关闭流资源

说明：程序中出现的异常需要使用**try-catch-finally**处理。

4.2 输出过程

- ① 创建**File**类的对象，指明写出的数据的位置。（不要求此文件一定要存在）
- ② 创建相应的输出流，将**File**类的对象作为参数，传入流的构造器中
- ③ 具体的写出过程：

write(char[]/byte[] buffer,0,len)

- ④ 关闭流资源

说明：程序中出现的异常需要使用**try-catch-finally**处理。

1.FileReader/FileWriter的使用：

1.1 FileReader的使用

```
/*
将day09 下的hello.txt文件内容读入程序中，并输出到控制台
```

说明点：

1. `read()`的理解：返回读入的一个字符。如果达到文件末尾，返回-1
2. 异常的处理：为了保证流资源一定可以执行关闭操作。需要使用try-catch-finally处理
3. 读入的文件一定要存在，否则就会报`FileNotFoundException`。

```
/*
@Test
public void testFileReader1() {
    FileReader fr = null;
    try {
        //1.File类的实例化
        File file = new File("hello.txt");

        //2.FileReader流的实例化
        fr = new FileReader(file);

        //3.读入的操作
        //read(char[] cbuf):返回每次读入cbuf数组中的字符的个数。如果达到文
件末尾，返回-1
        char[] cbuf = new char[5];
        int len;
        while((len = fr.read(cbuf)) != -1){
            //方式一：
            //错误的写法
            //for(int i = 0;i < cbuf.length;i++){
            //    System.out.print(cbuf[i]);
            //}
            //正确的写法
            for(int i = 0;i < len;i++){
                System.out.print(cbuf[i]);
            }
            //方式二：
            //错误的写法，对应着方式一的错误的写法
            String str = new String(cbuf);
            System.out.print(str);
            //正确的写法
            String str = new String(cbuf,0,len);
            System.out.print(str);
        }
    } catch (IOException e) {
        e.printStackTrace();
    } finally {
        if(fr != null){
            //4.资源的关闭
            try {
                fr.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

```

    }
}
```

1.2 FileWriter的使用

```
/*
从内存中写出数据到硬盘的文件里。
```

说明：

1. 输出操作，对应的File可以不存在的。并不会报异常
2. File对应的硬盘中的文件如果不存在，在输出的过程中，会自动创建此文件。
File对应的硬盘中的文件如果存在：
 如果流使用的构造器是：FileWriter(file, false) / FileWriter(file)：
对原文件的覆盖
 如果流使用的构造器是：FileWriter(file, true)：不会对原文件覆盖，而是在原文件基础上追加内容

```
/*
@Test
public void testFileWriter() {
    FileWriter fw = null;
    try {
        //1. 提供File类的对象，指明写出到的文件
        File file = new File("hello1.txt");

        //2. 提供FileWriter的对象，用于数据的写出
        fw = new FileWriter(file, false);

        //3. 写出的操作
        fw.write("I have a dream!\n");
        fw.write("you need to have a dream!");
    } catch (IOException e) {
        e.printStackTrace();
    } finally {
        //4. 流资源的关闭
        if(fw != null){
            try {
                fw.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

1.3 文本文件的复制：

```
@Test
public void testFileReaderFileWriter() {
    FileReader fr = null;
    FileWriter fw = null;
    try {
        //1. 创建File类的对象，指明读入和写出的文件
        File srcFile = new File("hello.txt");
        File destFile = new File("hello2.txt");

        //不能使用字符流来处理图片等字节数据
        //File srcFile = new File("爱情与友情.jpg");
        //File destFile = new File("爱情与友情1.jpg");
    }
```

```

//2. 创建输入流和输出流的对象
fr = new FileReader(srcFile);
fw = new FileWriter(destFile);

//3. 数据的读入和写出操作
char[] cbuf = new char[5];
int len; //记录每次读入到cbuf数组中的字符的个数
while((len = fr.read(cbuf)) != -1){
    //每次写出len个字符
    fw.write(cbuf, 0, len);

}

} catch (IOException e) {
    e.printStackTrace();
} finally {
    //4. 关闭流资源
    //方式一:
    //
    //    try {
    //        if(fw != null)
    //            fw.close();
    //    } catch (IOException e) {
    //        e.printStackTrace();
    //    }finally{
    //        try {
    //            if(fr != null)
    //                fr.close();
    //        } catch (IOException e) {
    //            e.printStackTrace();
    //        }
    //    }
    //方式二:
    try {
        if(fw != null)
            fw.close();
    } catch (IOException e) {
        e.printStackTrace();
    }

    try {
        if(fr != null)
            fr.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
}
}

```

2. FileInputStream / FileOutputStream的使用:

```

* 1. 对于文本文件(.txt,.java,.c,.cpp), 使用字符流处理
* 2. 对于非文本文件(.jpg,.mp3,.mp4,.avi,.doc,.ppt,...), 使用字节流处理
/*
实现对图片的复制操作
*/
@Test
public void testFileInputStreamOutputStream() {
    FileInputStream fis = null;

```

```

FileOutputStream fos = null;
try {
    //1. 造文件
    File srcFile = new File("爱情与友情.jpg");
    File destFile = new File("爱情与友情2.jpg");

    //2. 造流
    fis = new FileInputStream(srcFile);
    fos = new FileOutputStream(destFile);

    //3. 复制的过程
    byte[] buffer = new byte[5];
    int len;
    while((len = fis.read(buffer)) != -1){
        fos.write(buffer,0,len);
    }

} catch (IOException e) {
    e.printStackTrace();
} finally {
    if(fos != null){
        //4. 关闭流
        try {
            fos.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    if(fis != null){
        try {
            fis.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
}

```

【注意】

相对路径在IDEA和Eclipse中使用的区别？

IDEA：

如果使用单元测试方法，相对路径基于当前的Module的。

如果使用main()测试，相对路径基于当前Project的。

Eclipse：

单元测试方法还是main()，相对路径都是基于当前Project的。

缓冲流的使用

1. 缓冲流涉及到的类：

- * *BufferedInputStream*
- * *BufferedOutputStream*
- * *BufferedReader*
- * *BufferedWriter*

2. 作用：

作用：提供流的读取、写入的速度

提高读写速度的原因：内部提供了一个缓冲区。默认情况下是8kb

```
public
class BufferedInputStream extends FilterInputStream {

    private static int DEFAULT_BUFFER_SIZE = 8192;
```

3. 典型代码

3.1 使用BufferedInputStream和BufferedOutputStream：处理非文本文件

```
//实现文件复制的方法
public void copyFileWithBuffered(String srcPath,String destPath){
    BufferedInputStream bis = null;
    BufferedOutputStream bos = null;

    try {
        //1. 造文件
        File srcFile = new File(srcPath);
        File destFile = new File(destPath);
        //2. 造流
        //2.1 造节点流
        FileInputStream fis = new FileInputStream((srcFile));
        FileOutputStream fos = new FileOutputStream(destFile);
        //2.2 造缓冲流
        bis = new BufferedInputStream(fis);
        bos = new BufferedOutputStream(fos);

        //3. 复制的细节：读取、写入
        byte[] buffer = new byte[1024];
        int len;
        while((len = bis.read(buffer)) != -1){
            bos.write(buffer,0,len);
        }
    } catch (IOException e) {
        e.printStackTrace();
    } finally {
        //4. 资源关闭
        //要求：先关闭外层的流，再关闭内层的流
        if(bos != null){
            try {
                bos.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        if(bis != null){
            try {
                bis.close();
            }
        }
    }
}
```

```

        } catch (IOException e) {
            e.printStackTrace();
        }

    }
    //说明：关闭外层流的同时，内层流也会自动的进行关闭。关于内层流的关闭，我们可以省略。
    //    fos.close();
    //    fis.close();
}
}

```

3.2 使用BufferedReader和BufferedWriter：处理文本文件

```

@Test
public void testBufferedReaderBufferedWriter(){
    BufferedReader br = null;
    BufferedWriter bw = null;
    try {
        //创建文件和相应的流
        br = new BufferedReader(new FileReader(new File("dbcp.txt")));
        bw = new BufferedWriter(new FileWriter(new File
("dbcp1.txt")));

        //读写操作
        //方式一：使用char[]数组
        //        char[] cbuf = new char[1024];
        //        int len;
        //        while((len = br.read(cbuf)) != -1){
        //            bw.write(cbuf, 0, len);
        //            bw.flush();
        //        }

        //方式二：使用String
        String data;
        while((data = br.readLine()) != null){
            //方法一：
            //        bw.write(data + "\n");//data中不包含换行符
            //方法二：
            bw.write(data);//data中不包含换行符
            bw.newLine();//提供换行的操作

        }

    } catch (IOException e) {
        e.printStackTrace();
    } finally {
        //关闭资源
        if(bw != null){
            try {
                bw.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        if(br != null){
            try {
                br.close();
            }
        }
    }
}

```

```
    } catch (IOException e) {
        e.printStackTrace();
    }

}
```

转换流的使用

1. 转换流涉及到的类：属于字符流

InputStreamReader: 将一个字节的输入流转换为字符的输入流

解码：字节、字节数组 ---> 字符数组、字符串

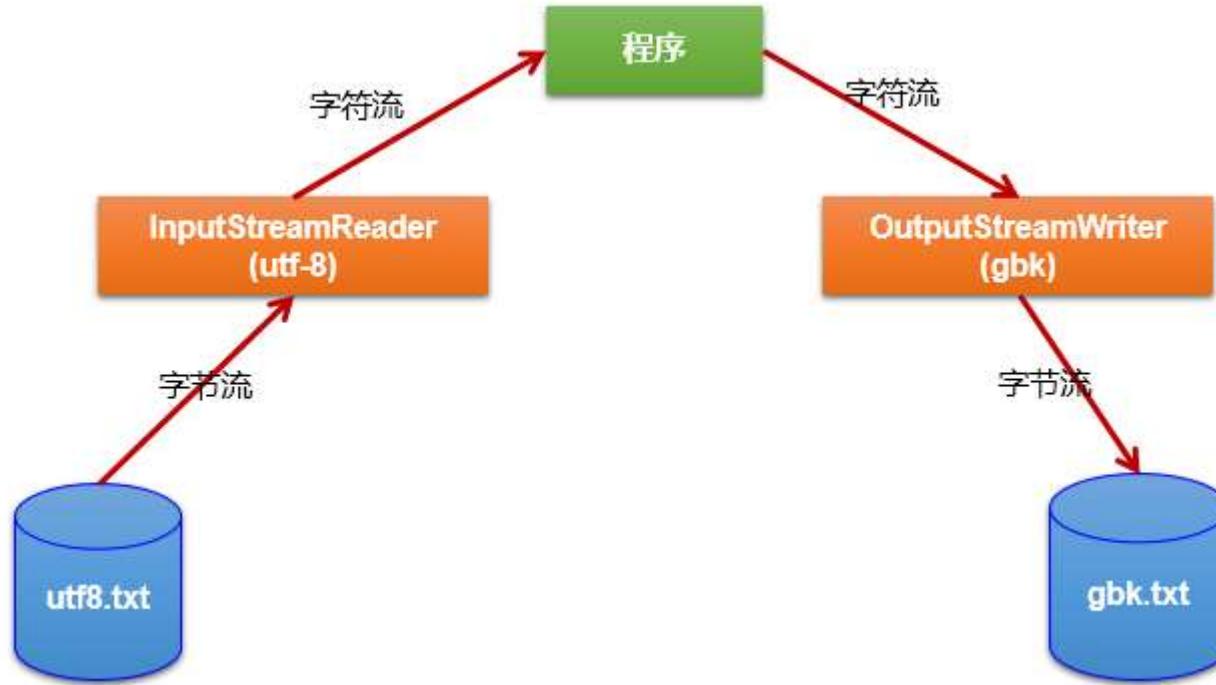
OutputStreamWriter: 将一个字符的输出流转换为字节的输出流

编码：字符数组、字符串 ---> 字节、字节数组

说明：编码决定了解码的方式

2. 作用：提供字节流与字符流之间的转换

3. 图示：



4. 典型实现：

```

@Test
public void test1() throws IOException {
    FileInputStream fis = new FileInputStream("dbcp.txt");
    // InputStreamReader isr = new InputStreamReader(fis); // 使用系统默认的字符集
    // 参数2指明了字符集，具体使用哪个字符集，取决于文件dbcp.txt保存时使用的字符集
    InputStreamReader isr = new InputStreamReader(fis, "UTF-8"); // 使用系统默认的字符集

    char[] cbuf = new char[20];
    int len;
    while((len = isr.read(cbuf)) != -1){
        String str = new String(cbuf, 0, len);
        System.out.print(str);
    }

    isr.close();
}

/*
此时处理异常的话，仍然应该使用try-catch-finally
综合使用InputStreamReader和OutputStreamWriter
*/
@Test
public void test2() throws Exception {
    //1. 造文件、造流
}
  
```

```
File file1 = new File("dbcp.txt");
File file2 = new File("dbcp_gbk.txt");

FileInputStream fis = new FileInputStream(file1);
FileOutputStream fos = new FileOutputStream(file2);

InputStreamReader isr = new InputStreamReader(fis,"utf-8");
OutputStreamWriter osw = new OutputStreamWriter(fos,"gbk");

//2. 读写过程
char[] cbuf = new char[20];
int len;
while((len = isr.read(cbuf)) != -1){
    osw.write(cbuf,0,len);
}

//3. 关闭资源
isr.close();
osw.close();

}

5. 说明：
//文件编码的方式（比如：GBK），决定了解析时使用的字符集（也只能是GBK）。
```

1. 常见的编码表

- ⌚ **ASCII:** 美国标准信息交换码。
⌚ 用一个字节的7位可以表示。
- ⌚ **ISO8859-1:** 拉丁码表。欧洲码表
⌚ 用一个字节的8位表示。
- ⌚ **GB2312:** 中国的中文编码表。最多两个字节编码所有字符
- ⌚ **GBK:** 中国的中文编码表升级，融合了更多的中文文字符号。最多两个字节编码
- ⌚ **Unicode:** 国际标准码，融合了目前人类使用的所字符。为每个字符分配唯一的字符码。所有的文字都用两个字节来表示。
- ⌚ **UTF-8:** 变长的编码方式，可用1-4个字节来表示一个字符。

2. 对后面学习的启示

客户端/浏览器端 <----> 后台(java, Go, Python, Node.js, PHP) <----> 数据库

要求前前后后使用的字符集都要统一：UTF-8.

1. 标准的输入输出流:

`System.in`: 标准的输入流, 默认从键盘输入

`System.out`: 标准的输出流, 默认从控制台输出

修改默认的输入和输出行为:

`System`类的`setIn(InputStream is)` / `setOut(PrintStream ps)`方式重新指定输入和输出的流。

2. 打印流:

`PrintStream` 和 `PrintWriter`

说明:

- ① 提供了一系列重载的`print()`和`println()`方法, 用于多种数据类型的输出
- ② `System.out`返回的是`PrintStream`的实例

3. 数据流:

`DataInputStream` 和 `DataOutputStream`

作用:

用于读取或写出基本数据类型的变量或字符串

示例代码:

```
/*
练习: 将内存中的字符串、基本数据类型的变量写出到文件中。

注意: 处理异常的话, 仍然应该使用try-catch-finally.

*/
@Test
public void test3() throws IOException {
    //1.
    DataOutputStream dos = new DataOutputStream(new FileOutputStream("data.txt"));
    //2.
    dos.writeUTF("刘建辰");
    dos.flush(); //刷新操作, 将内存中的数据写入文件
    dos.writeInt(23);
    dos.flush();
    dos.writeBoolean(true);
    dos.flush();
    //3.
    dos.close();
}
```

将文件中存储的基本数据类型变量和字符串读取到内存中, 保存在变量中。

注意点: 读取不同类型的数据的顺序要与当初写入文件时, 保存的数据的顺序一致!

```
/*
*/
@Test
public void test4() throws IOException {
    //1.
    DataInputStream dis = new DataInputStream(new FileInputStream("data.txt"));
    //2.
    String name = dis.readUTF();
```

```
int age = dis.readInt();
boolean isMale = dis.readBoolean();

System.out.println("name = " + name);
System.out.println("age = " + age);
System.out.println("isMale = " + isMale);

//3.
dis.close();

}
```

对象流的使用

1. 对象流：

`ObjectInputStream` 和 `ObjectOutputStream`

2. 作用：

`ObjectOutputStream`: 内存中的对象--->存储中的文件、通过网络传输出去：序列化过程

`ObjectInputStream`: 存储中的文件、通过网络接收过来 --->内存中的对象：反序列化过程

3. 对象的序列化机制：

对象序列化机制允许把内存中的Java对象转换成平台无关的二进制流，从而允许把这种二进制流持久地保存在磁盘上，或通过网络将这种二进制流传输到另一个网络节点。//当其它程序获取了这种二进制流，就可以恢复成原来的Java对象

4.

序列化代码实现：

```

@Test
public void testObjectOutputStream(){
    ObjectOutputStream oos = null;

    try {
        //1.
        oos = new ObjectOutputStream(new FileOutputStream("object.dat"));
        //2.
        oos.writeObject(new String("我爱北京天安门"));
        oos.flush(); //刷新操作

        oos.writeObject(new Person("王铭", 23));
        oos.flush();

        oos.writeObject(new Person("张学良", 23, 1001, new Account(5000)));
        oos.flush();

    } catch (IOException e) {
        e.printStackTrace();
    } finally {
        if(oos != null){
            //3.
            try {
                oos.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}

```

反序列化代码实现：

```

@Test
public void testObjectInputStream(){
    ObjectInputStream ois = null;
    try {
        ois = new ObjectInputStream(new FileInputStream("object.dat"));

        Object obj = ois.readObject();
        String str = (String) obj;

        Person p = (Person) ois.readObject();
        Person p1 = (Person) ois.readObject();

        System.out.println(str);
        System.out.println(p);
        System.out.println(p1);

    } catch (IOException e) {
        e.printStackTrace();
    } catch (ClassNotFoundException e) {
        e.printStackTrace();
    } finally {
        if(ois != null){
            try {
                ois.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}

```

}

6. 实现序列化的对象所属的类需要满足：

1. 需要实现接口: `Serializable`
- * 2. 当前类提供一个全局常量: `serialVersionUID`
- * 3. 除了当前`Person`类需要实现`Serializable`接口之外，还必须保证其内部所属性也必须是可序列化的。（默认情况下，基本数据类型可序列化）
- *
- *
- * 补充: `ObjectOutputStream`和 `ObjectInputStream`不能序列化`static`和`transient`修饰的成员变量
- *

RandomAccessFile的使用

1. 随机存取文件流： RandomAccessFile

2. 使用说明：

- * 1. RandomAccessFile直接继承于java.lang.Object类，实现了DataInput和DataOutput接口
- * 2. RandomAccessFile既可以作为一个输入流，又可以作为一个输出流
- *
- * 3. 如果RandomAccessFile作为输出流时，写出到的文件如果不存在，则在执行过程中自动创建。
- * 如果写出到的文件存在，则会对原文件内容进行覆盖。（默认情况下，从头覆盖）
- *
- * 4. 可以通过相关的操作，实现RandomAccessFile“插入”数据的效果。seek(int pos)

3.

典型代码1：

```

@Test
public void test1() {

    RandomAccessFile raf1 = null;
    RandomAccessFile raf2 = null;
    try {
        //1.
        raf1 = new RandomAccessFile(new File("爱情与友情.jpg"), "r");
        raf2 = new RandomAccessFile(new File("爱情与友情1.jpg"), "rw");
        //2.
        byte[] buffer = new byte[1024];
        int len;
        while((len = raf1.read(buffer)) != -1){
            raf2.write(buffer, 0, len);
        }
    } catch (IOException e) {
        e.printStackTrace();
    } finally {
        //3.
        if(raf1 != null){
            try {
                raf1.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        if(raf2 != null){
            try {
                raf2.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}

```

典型代码2：

```

/*
使用RandomAccessFile实现数据的插入效果
*/

```

```
@Test
public void test3() throws IOException {
    RandomAccessFile raf1 = new RandomAccessFile("hello.txt", "rw");

    raf1.seek(3); // 将指针调到角标为3的位置
    // 保存指针3后面的所数据到StringBuilder中
    StringBuilder builder = new StringBuilder((int) new File
("hello.txt").length());
    byte[] buffer = new byte[20];
    int len;
    while ((len = raf1.read(buffer)) != -1) {
        builder.append(new String(buffer, 0, len));
    }
    // 调回指针，写入“xyz”
    raf1.seek(3);
    raf1.write("xyz".getBytes());

    // 将StringBuilder中的数据写入到文件中
    raf1.write(builder.toString().getBytes());

    raf1.close();
}

// 思考：将StringBuilder替换为ByteArrayOutputStream
}
```

1.NIO的使用说明：

>Java NIO (New IO, Non-Blocking IO)是从Java 1.4版本开始引入的一套新的IO API，可以替代标准的Java IO API。

>NIO与原来的IO同样的作用和目的，但是使用的方式完全不同，NIO支持面向缓冲区的（IO是面向流的）、基于通道的IO操作。

>**NIO将以更加高效的方式进行文件的读写操作。**

>随着JDK 7的发布，Java对NIO进行了极大的扩展，增强了对文件处理和文件系统特性的支持，以至于我们称他们为 NIO.2。

2.Path的使用 ---jdk7提供

2.1Path的说明：

Path替换原有的File类。

2.2如何实例化：

- Paths 类提供的静态 get() 方法用来获取 Path 对象：

- static Path get(String first, String ... more) : 用于将多个字符串串连成路径
- static Path get(URI uri) : 返回指定uri对应的Path路径

2.3常用方法：

- String toString() : 返回调用 Path 对象的字符串表示形式
- boolean startsWith(String path) : 判断是否以 path 路径开始
- boolean endsWith(String path) : 判断是否以 path 路径结束
- boolean isAbsolute() : 判断是否是绝对路径
- Path getParent() : 返回Path对象包含整个路径，不包含 Path 对象指定的文件路径
- Path getRoot() : 返回调用 Path 对象的根路径
- Path getFileName() : 返回与调用 Path 对象关联的文件名
- int getNameCount() : 返回Path 根目录后面元素的数量
- Path getName(int idx) : 返回指定索引位置 idx 的路径名称
- Path toAbsolutePath() : 作为绝对路径返回调用 Path 对象
- Path resolve(Path p) : 合并两个路径，返回合并后的路径对应的Path对象
- File toFile() : 将Path转化为File类的对象

3.Files工具类 ---jdk7提供

3.1作用：

操作文件或文件目录的工具类

3.2常用方法：

- `Path copy(Path src, Path dest, CopyOption ... how)` : 文件的复制
- `Path createDirectory(Path path, FileAttribute<?> ... attr)` : 创建一个目录
- `Path createFile(Path path, FileAttribute<?> ... arr)` : 创建一个文件
- `void delete(Path path)` : 删除一个文件/目录, 如果不存在, 执行报错
- `void deleteIfExists(Path path)` : Path对应的文件/目录如果存在, 执行删除
- `Path move(Path src, Path dest, CopyOption...how)` : 将 src 移动到 dest 位置
- `long size(Path path)` : 返回 path 指定文件的大小

Files常用方法: 用于判断

- `boolean exists(Path path, LinkOption ... opts)` : 判断文件是否存在
- `boolean isDirectory(Path path, LinkOption ... opts)` : 判断是否是目录
- `boolean isRegularFile(Path path, LinkOption ... opts)` : 判断是否是文件
- `boolean isHidden(Path path)` : 判断是否是隐藏文件
- `boolean isReadable(Path path)` : 判断文件是否可读
- `boolean isWritable(Path path)` : 判断文件是否可写
- `boolean notExists(Path path, LinkOption ... opts)` : 判断文件是否不存在

Files常用方法: 用于操作内容

- `SeekableByteChannel newByteChannel(Path path, OpenOption...how)` : 获取与指定文件的连接, how 指定打开方式。
- `DirectoryStream<Path> newDirectoryStream(Path path)` : 打开 path 指定的目录
- `InputStream newInputStream(Path path, OpenOption...how)` : 获取 InputStream 对象
- `OutputStream newOutputStream(Path path, OpenOption...how)` : 获取 OutputStream 对象

InetAddress类的使用

一、实现网络通信需要解决的两个问题

- * 1. 如何准确地定位网络上一台或多台主机；定位主机上的特定的应用
- * 2. 找到主机后如何可靠高效地进行数据传输

二、网络通信的两个要素：

- * 1. 对应问题一：IP和端口号
- * 2. 对应问题二：提供网络通信协议：TCP/IP参考模型（应用层、传输层、网络层、物理+数据链路层）

三、通信要素一：IP和端口号

1. IP的理解

- * 1. IP：唯一的标识 Internet 上的计算机（通信实体）
- * 2. 在Java中使用InetAddress类代表IP
- * 3. IP分类：IPv4 和 IPv6；万维网 和 局域网
- * 4. 域名： www.baidu.com www.mi.com www.sina.com www.jd.com
- *

域名解析：域名容易记忆，当在连接网络时输入一个主机的域名后，域名服务器(DNS)负责将域名转化成IP地址，这样才能和主机建立连接。 -----域名解析

- * 5. 本地回路地址：127.0.0.1 对应着：localhost
- *

2. InetAddress类：此类的一个对象就代表着一个具体的IP地址

2.1实例化

`getByName(String host)` 、 `getLocalHost()`

2.2常用方法

`getHostName() / getHostAddress()`

3. 端口号：正在计算机上运行的进程。

- * 要求：不同的进程不同的端口号
- * 范围：被规定为一个 16 位的整数 0~65535。

端口号与IP地址的组合得出一个网络套接字：Socket

四、通信要素二：网络通信协议

1. 分型模型

OSI参考模型	TCP/IP参考模型	TCP/IP参考模型各层对应协议
应用层		
表示层	应用层	HTTP、FTP、Telnet、DNS...
会话层		
传输层	传输层	TCP、UDP、...
网络层	网络层	IP、ICMP、ARP...
数据链路层		
物理层	物理+数据链路层	Link

2. TCP和UDP的区别

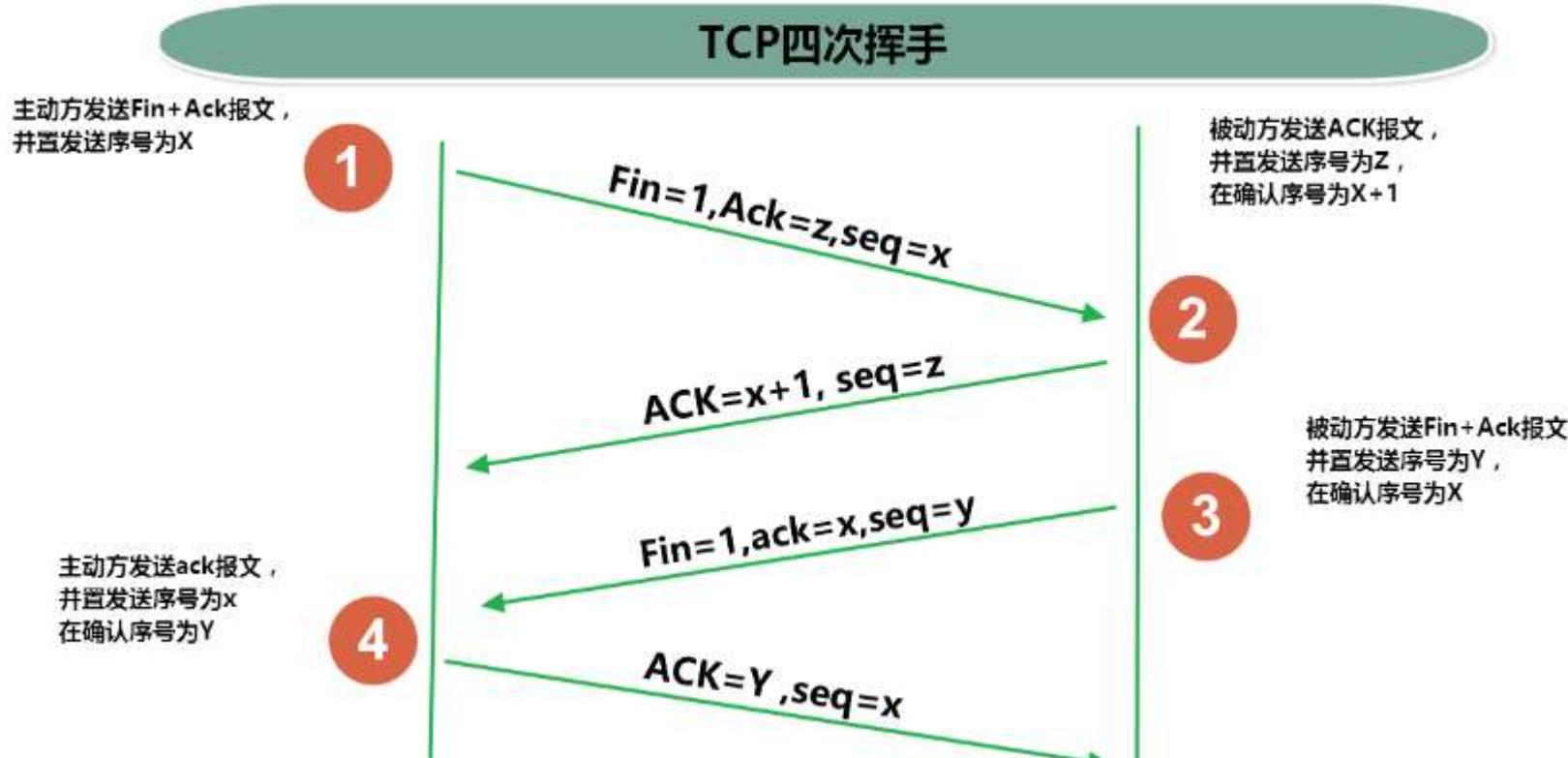
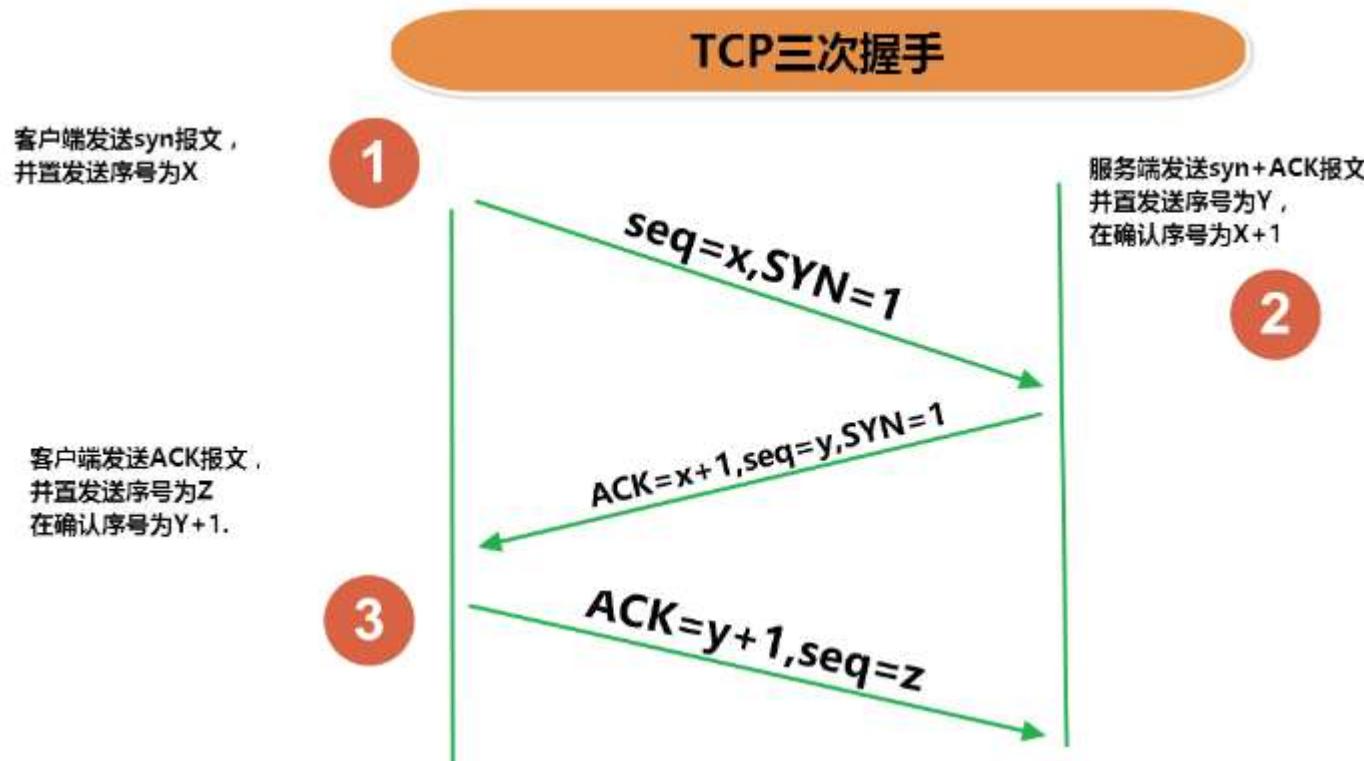
● TCP协议：

- 使用TCP协议前，须先建立TCP连接，形成传输数据通道
- 传输前，采用“三次握手”方式，点对点通信，是可靠的
- TCP协议进行通信的两个应用进程：客户端、服务端。
- 在连接中可进行大数据量的传输
- 传输完毕，需释放已建立的连接，效率低

● UDP协议：

- 将数据、源、目的封装成数据包，不需要建立连接
- 每个数据报的大小限制在64K内
- 发送不管对方是否准备好，接收方收到也不确认，故是不可靠的
- 可以广播发送
- 发送数据结束时无需释放资源，开销小，速度快

3. TCP三次握手和四次挥手



代码示例1：客户端发送信息给服务端，服务端将数据显示在控制台上

```
//客户端
@Test
public void client() {
    Socket socket = null;
    OutputStream os = null;
    try {
        //1. 创建Socket对象，指明服务器端的ip和端口号
        InetAddress inet = InetAddress.getByName("192.168.14.100");
        socket = new Socket(inet, 8899);
        //2. 获取一个输出流，用于输出数据
        os = socket.getOutputStream();
        //3. 写出数据的操作
        os.write("你好，我是客户端mm".getBytes());
    } catch (IOException e) {
        e.printStackTrace();
    } finally {
        //4. 资源的关闭
        if(os != null){
            try {
                os.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        if(socket != null){
            try {
                socket.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}

//服务端
@Test
public void server() {
    ServerSocket ss = null;
    Socket socket = null;
    InputStream is = null;
    ByteArrayOutputStream baos = null;
    try {
        //1. 创建服务器端的ServerSocket，指明自己的端口号
        ss = new ServerSocket(8899);
        //2. 调用accept()表示接收来自于客户端的socket
        socket = ss.accept();
        //3. 获取输入流
        is = socket.getInputStream();

        //不建议这样写，可能会乱码
        byte[] buffer = new byte[1024];
        int len;
        while((len = is.read(buffer)) != -1){
            String str = new String(buffer, 0, len);
        }
    } catch (IOException e) {
        e.printStackTrace();
    } finally {
        if(ss != null){
            try {
                ss.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        if(socket != null){
            try {
                socket.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        if(is != null){
            try {
                is.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        if(baos != null){
            try {
                baos.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

```

//           System.out.print(str);
//       }
//4. 读取输入流中的数据
baos = new ByteArrayOutputStream();
byte[] buffer = new byte[5];
int len;
while((len = is.read(buffer)) != -1){
    baos.write(buffer,0,len);
}

System.out.println(baos.toString());

System.out.println("收到了来自于: " + socket.getInetAddress()
().getHostAddress() + "的数据");

} catch (IOException e) {
    e.printStackTrace();
} finally {
    if(baos != null){
        //5. 关闭资源
        try {
            baos.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    if(is != null){
        try {
            is.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    if(socket != null){
        try {
            socket.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    if(ss != null){
        try {
            ss.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
}
}

```

代码示例2：客户端发送文件给服务端，服务端将文件保存在本地。

```

/*
这里涉及到的异常，应该使用try-catch-finally处理
*/
@Test
public void client() throws IOException {
//1.
Socket socket = new Socket(InetAddress.getByName("127.0.0.1"),9090);
//2.
OutputStream os = socket.getOutputStream();

```

```

//3.
FileInputStream fis = new FileInputStream(new File("beauty.jpg"));
//4.
byte[] buffer = new byte[1024];
int len;
while((len = fis.read(buffer)) != -1){
    os.write(buffer,0,len);
}
//5.
fis.close();
os.close();
socket.close();
}

/*
这里涉及到的异常，应该使用try-catch-finally处理
*/
@Test
public void server() throws IOException {
    //1.
    ServerSocket ss = new ServerSocket(9090);
    //2.
    Socket socket = ss.accept();
    //3.
    InputStream is = socket.getInputStream();
    //4.
    FileOutputStream fos = new FileOutputStream(new File("beauty1.jpg"));
    //5.
    byte[] buffer = new byte[1024];
    int len;
    while((len = is.read(buffer)) != -1){
        fos.write(buffer,0,len);
    }
    //6.
    fos.close();
    is.close();
    socket.close();
    ss.close();
}

}

```

代码示例3：从客户端发送文件给服务端，服务端保存到本地。并返回“发送成功”给客户端。并关闭相应的连接。

```

/*
这里涉及到的异常，应该使用try-catch-finally处理
*/
@Test
public void client() throws IOException {
    //1.
    Socket socket = new Socket(InetAddress.getByName("127.0.0.1"),9090);
    //2.
    OutputStream os = socket.getOutputStream();
    //3.
    FileInputStream fis = new FileInputStream(new File("beauty.jpg"));
    //4.
    byte[] buffer = new byte[1024];
    int len;
    while((len = fis.read(buffer)) != -1){
        os.write(buffer,0,len);
    }
    //关闭数据的输出
}

```

```

socket.shutdownOutput();

//5. 接收来自于服务器端的数据，并显示到控制台上
InputStream is = socket.getInputStream();
ByteArrayOutputStream baos = new ByteArrayOutputStream();
byte[] bufferr = new byte[20];
int len1;
while((len1 = is.read(buffer)) != -1){
    baos.write(buffer, 0, len1);
}

System.out.println(baos.toString());

//6.
fis.close();
os.close();
socket.close();
baos.close();
}

/*
这里涉及到的异常，应该使用try-catch-finally处理
*/
@Test
public void server() throws IOException {
    //1.
    ServerSocket ss = new ServerSocket(9090);
    //2.
    Socket socket = ss.accept();
    //3.
    InputStream is = socket.getInputStream();
    //4.
    FileOutputStream fos = new FileOutputStream(new File("beauty2.jpg"));
    //5.
    byte[] buffer = new byte[1024];
    int len;
    while((len = is.read(buffer)) != -1){
        fos.write(buffer, 0, len);
    }

    System.out.println("图片传输完成");

    //6. 服务器端给予客户端反馈
    OutputStream os = socket.getOutputStream();
    os.write("你好，美女，照片我已收到，非常漂亮!".getBytes());

    //7.
    fos.close();
    is.close();
    socket.close();
    ss.close();
    os.close();
}

```

代码示例：

```
//发送端
@Test
public void sender() throws IOException {
    DatagramSocket socket = new DatagramSocket();
    String str = "我是UDP方式发送的导弹";
    byte[] data = str.getBytes();
    InetAddress inet = InetAddress.getLocalHost();
    DatagramPacket packet = new DatagramPacket(data,0,data.length,inet,9090);
    socket.send(packet);
    socket.close();
}

//接收端
@Test
public void receiver() throws IOException {
    DatagramSocket socket = new DatagramSocket(9090);
    byte[] buffer = new byte[100];
    DatagramPacket packet = new DatagramPacket(buffer,0,buffer.length);
    socket.receive(packet);
    System.out.println(new String(packet.getData(),0,packet.getLength()));
    socket.close();
}
```

1. URL(Uniform Resource Locator)的理解：
统一资源定位符，对应着互联网的某一资源地址

2. URL的5个基本结构：

* `http://localhost:8080/examples/beauty.jpg?username=Tom`
 * 协议 主机名 端口号 资源地址 参数列表

3. 如何实例化：

```
URL url = new URL("http://localhost:8080/examples/beauty.jpg?  
username=Tom");
```

4. 常用方法：

- `public String getProtocol()` 获取该URL的协议名
- `public String getHost()` 获取该URL的主机名
- `public String getPort()` 获取该URL的端口号
- `public String getPath()` 获取该URL的文件路径
- `public String getFile()` 获取该URL的文件名
- `public String getQuery()` 获取该URL的查询名

5. 可以读取、下载对应的url资源：

```
public static void main(String[] args) {  
  
    HttpURLConnection urlConnection = null;  
    InputStream is = null;  
    FileOutputStream fos = null;  
    try {  
        URL url = new URL("http://localhost:8080/examples/beauty.jpg");  
  
        urlConnection = (HttpURLConnection) url.openConnection();  
  
        urlConnection.connect();  
  
        is = urlConnection.getInputStream();  
        fos = new FileOutputStream("day10\\beauty3.jpg");  
  
        byte[] buffer = new byte[1024];  
        int len;  
        while((len = is.read(buffer)) != -1){  
            fos.write(buffer, 0, len);  
        }  
  
        System.out.println("下载完成");  
    } catch (IOException e) {  
        e.printStackTrace();  
    } finally {  
        //关闭资源  
        if(is != null){  
            try {  
                is.close();  
            } catch (IOException e) {  
                e.printStackTrace();  
            }  
        }  
        if(fos != null){  
            try {  
                fos.close();  
            } catch (IOException e) {  
                e.printStackTrace();  
            }  
        }  
    }  
}
```

```
        e.printStackTrace();
    }
}
if(urlConnection != null){
    urlConnection.disconnect();
}
}
```

1. 本章的主要内容

- 1 Java反射机制概述**
- 2 理解Class类并获取Class实例**
- 3 类的加载与ClassLoader的理解**
- 4 创建运行时类的对象**
- 5 获取运行时类的完整结构**
- 6 调用运行时类的指定结构**
- 7 反射的应用：动态代理**



2. 关于反射的理解

Reflection (反射) 是被视为**动态语言**的关键，反射机制允许程序在执行期借助于**Reflection API**取得任何类的内部信息，并能直接操作任意对象的内部属性及方法。

框架 = 反射 + 注解 + 设计模式。

3. 体会反射机制的“动态性”

```
// 体会反射的动态性
@Test
public void test2(){

    for(int i = 0;i < 100;i++){
        int num = new Random().nextInt(3); //0,1,2
        String classPath = "";
        switch(num){
            case 0:
                classPath = "java.util.Date";
                break;
            case 1:
                classPath = "java.lang.Object";
                break;
            case 2:
                classPath = "com.atguigu.java.Person";
                break;
        }

        try {
            Object obj = getInstance(classPath);
            System.out.println(obj);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

```
}

/*
创建一个指定类的对象。
classPath: 指定类的全类名
*/
public Object getInstance(String classPath) throws Exception {
    Class clazz = Class.forName(classPath);
    return clazz.newInstance();
}
```

4. 反射机制能提供的功能

- 在运行时判断任意一个对象所属的类
- 在运行时构造任意一个类的对象
- 在运行时判断任意一个类所具有的成员变量和方法
- 在运行时获取泛型信息
- 在运行时调用任意一个对象的成员变量和方法
- 在运行时处理注解
- 生成动态代理

5. 相关API

java.lang.Class: 反射的源头
java.lang.reflect.Method
java.lang.reflect.Field
java.lang.reflect.Constructor
....

Class类的理解与获取Class的实例

1. Class类的理解

1. 类的加载过程:

程序经过javac.exe命令以后，会生成一个或多个字节码文件(.class结尾)。

接着我们使用java.exe命令对某个字节码文件进行解释运行。相当于将某个字节码文件加载到内存中。此过程就称为类的加载。加载到内存中的类，我们就称为运行时类，此运行时类，就作为Class的一个实例。

2. 换句话说，Class的实例就对应着一个运行时类。

3. 加载到内存中的运行时类，会缓存一定的时间。在此时间之内，我们可以通过不同的方式来获取此运行时类。

2. 获取Class实例的几种方式：（前三种方式需要掌握）

```
// 方式一：调用运行时类的属性：.class
Class clazz1 = Person.class;
System.out.println(clazz1);

// 方式二：通过运行时类的对象，调用getClass()
Person p1 = new Person();
Class clazz2 = p1.getClass();
System.out.println(clazz2);

// 方式三：调用Class的静态方法：forName(String className)
Class clazz3 = Class.forName("com.atguigu.java.Person");
// clazz3 = Class.forName("java.lang.String");
System.out.println(clazz3);

System.out.println(clazz1 == clazz2);
System.out.println(clazz1 == clazz3);

// 方式四：使用类的加载器：ClassLoader (了解)
ClassLoader classLoader = ReflectionTest.class.getClassLoader();
Class clazz4 = classLoader.loadClass("com.atguigu.java.Person");
System.out.println(clazz4);

System.out.println(clazz1 == clazz4);
```

3. 总结：创建类的对象的方式？

方式一：new + 构造器

方式二：要创建Xxx类的对象，可以考虑：Xxx、Xxss、XxxFactory、XxxBuilder类中查看是否有

静态方法的存在。可以调用其静态方法，创建Xxx对象。

方式三：通过反射

4. Class实例可以是哪些结构的说明

(1) **class**:

外部类, 成员(成员内部类, 静态内部类), 局部内部类, 匿名内部类

(2) **interface**: 接口

(3) []: 数组

(4) **enum**: 枚举

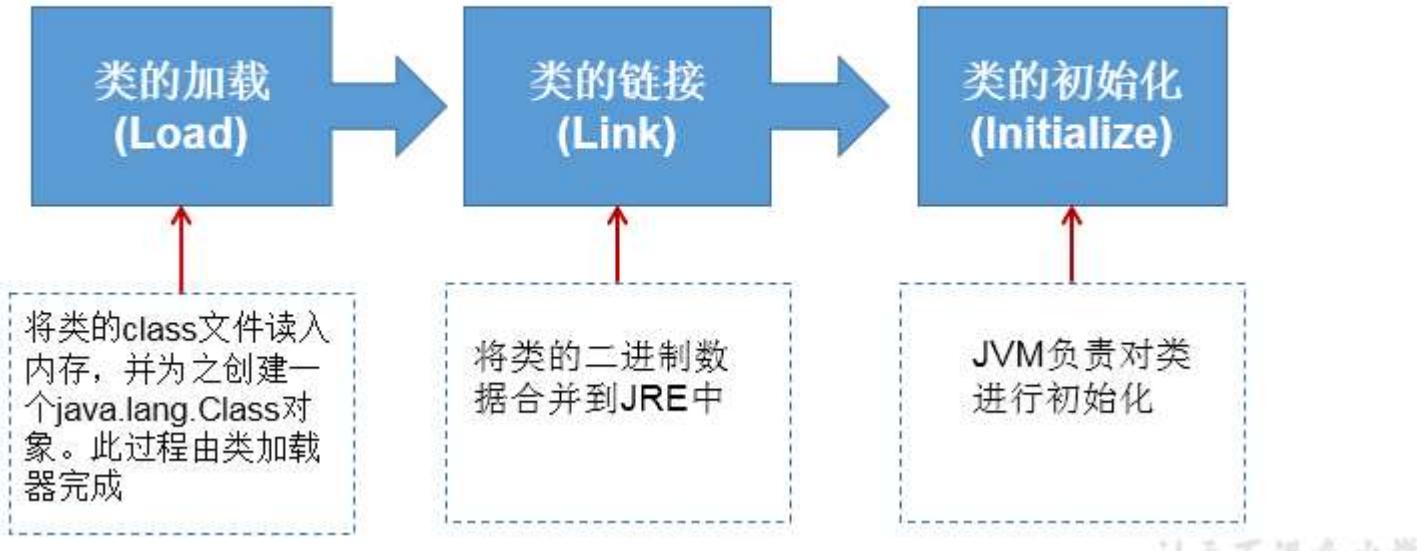
(5) **annotation**: 注解@**interface**

(6) **primitive type**: 基本数据类型

(7) **void**

1. 类的加载过程----了解

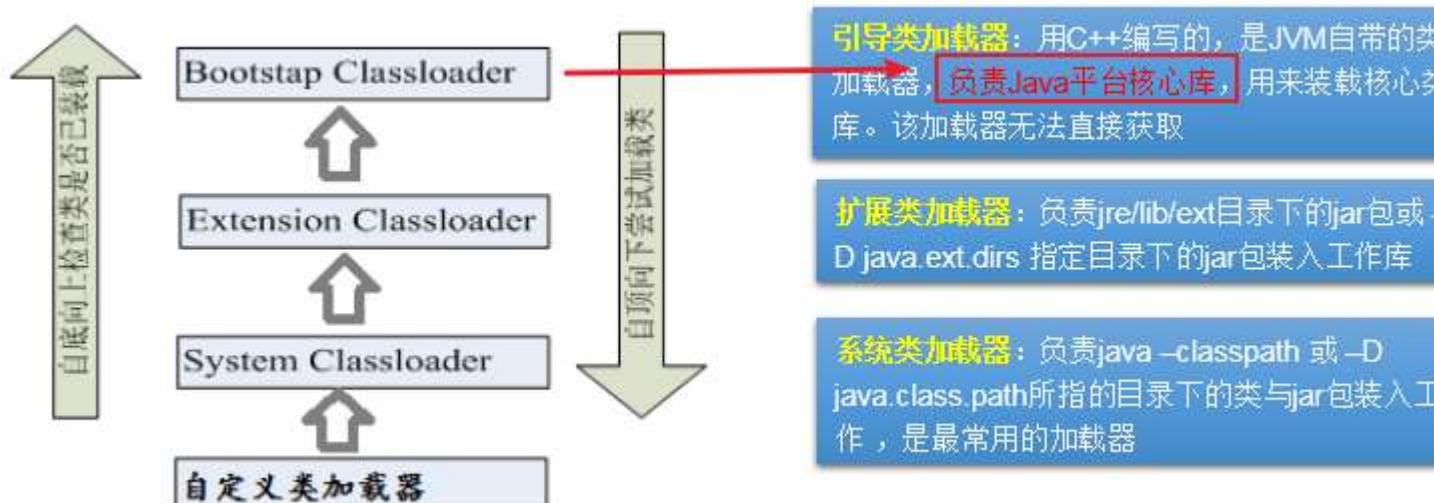
当程序主动使用某个类时，如果该类还未被加载到内存中，则系统会通过如下三个步骤来对该类进行初始化。



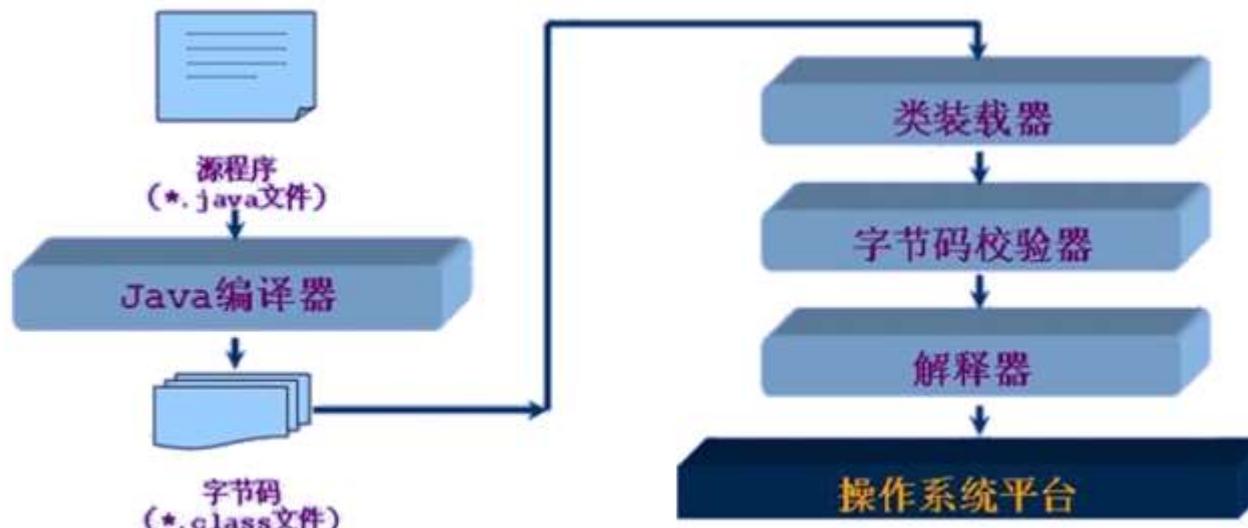
2. 类的加载器的作用

- 类加载的作用：**将class文件字节码内容加载到内存中，并将这些静态数据转换成方法区的运行时数据结构，然后在堆中生成一个代表这个类的java.lang.Class对象，作为方法区中类数据的访问入口。
- 类缓存：**标准的JavaSE类加载器可以按要求查找类，但一旦某个类被加载到类加载器中，它将维持加载（缓存）一段时间。不过JVM垃圾回收机制可以回收这些Class对象。

3. 类的加载器的分类



4. Java类编译、运行的执行的流程



5. 使用ClassLoader加载src目录下的配置文件

```
@Test
public void test2() throws Exception {
    Properties pros = new Properties();
    //此时的文件默认在当前的module下。
    //读取配置文件的方式一:
//    FileInputStream fis = new FileInputStream("jdbc.properties");
//    FileInputStream fis = new FileInputStream("src\
//jdbc1.properties");
//    pros.load(fis);

    //读取配置文件的方式二: 使用ClassLoader
    //配置文件默认识别为: 当前module的src下
    ClassLoader classLoader = ClassLoaderTest.class.getClassLoader();
    InputStream is = classLoader.getResourceAsStream(
    "jdbc1.properties");
    pros.load(is);

    String user = pros.getProperty("user");
    String password = pros.getProperty("password");
    System.out.println("user = " + user + ",password = " + password);
}
```

反射应用一：创建运行时类的对象

1. 代码举例

```
Class<Person> clazz = Person.class;  
  
Person obj = clazz.newInstance();  
System.out.println(obj);
```

2. 说明

`newInstance()`: 调用此方法，创建对应的运行时类的对象。内部调用了运行时类的空参的构造器。

要想此方法正常的创建运行时类的对象，要求：

1. 运行时类必须提供空参的构造器
2. 空参的构造器的访问权限得够。通常，设置为`public`。

在javabean中要求提供一个`public`的空参构造器。原因：

1. 便于通过反射，创建运行时类的对象
2. 便于子类继承此运行时类时，默认调用`super()`时，保证父类此构造器

反射应用二：获取运行时类的完整结构

我们可以通过反射，获取对应的运行时类中所有的属性、方法、构造器、父类、接口、父类的泛型、包、注解、异常等。。。

典型代码：

```

@Test
public void test1(){

    Class clazz = Person.class;

    // 获取属性结构
    //getFields(): 获取当前运行时类及其父类中声明为public访问权限的属性
    Field[] fields = clazz.getFields();
    for(Field f : fields){
        System.out.println(f);
    }
    System.out.println();

    //getDeclaredFields(): 获取当前运行时类中声明的所有属性。 (不包含父类中声明的属性)
    Field[] declaredFields = clazz.getDeclaredFields();
    for(Field f : declaredFields){
        System.out.println(f);
    }
}

@Test
public void test1(){

    Class clazz = Person.class;

    //getMethods(): 获取当前运行时类及其父类中声明为public权限的方法
    Method[] methods = clazz.getMethods();
    for(Method m : methods){
        System.out.println(m);
    }
    System.out.println();
    //getDeclaredMethods(): 获取当前运行时类中声明的所有方法。 (不包含父类中声明的方法)
    Method[] declaredMethods = clazz.getDeclaredMethods();
    for(Method m : declaredMethods){
        System.out.println(m);
    }
}

/*
    获取构造器结构

*/
@Test
public void test1(){

    Class clazz = Person.class;
    //getConstructors(): 获取当前运行时类中声明为public的构造器
    Constructor[] constructors = clazz.getConstructors();
    for(Constructor c : constructors){
        System.out.println(c);
    }
}

```

```

System.out.println();
//getDeclaredConstructors(): 获取当前运行时类中声明的所有的构造器
Constructor[] declaredConstructors = clazz.getDeclaredConstructors()
();
for(Constructor c : declaredConstructors){
    System.out.println(c);
}

/*
获取运行时类的父类

*/
@Test
public void test2(){
    Class clazz = Person.class;

    Class superclass = clazz.getSuperclass();
    System.out.println(superclass);
}

/*
获取运行时类的带泛型的父类

*/
@Test
public void test3(){
    Class clazz = Person.class;

    Type genericSuperclass = clazz.getGenericSuperclass();
    System.out.println(genericSuperclass);
}

/*
获取运行时类的带泛型的父类的泛型

代码：逻辑性代码 vs 功能性代码
*/
@Test
public void test4(){
    Class clazz = Person.class;

    Type genericSuperclass = clazz.getGenericSuperclass();
    ParameterizedType paramType = (ParameterizedType)
genericSuperclass;
    // 获取泛型类型
    Type[] actualTypeArguments = paramType.getActualTypeArguments();
//    System.out.println(actualTypeArguments[0].getTypeName());
    System.out.println(((Class)actualTypeArguments[0]).getName());
}

/*
获取运行时类实现的接口
*/
@Test
public void test5(){
    Class clazz = Person.class;

    Class[] interfaces = clazz.getInterfaces();
    for(Class c : interfaces){
        System.out.println(c);
}

```

```
}

System.out.println();
//获取运行时类的父类实现的接口
Class[] interfaces1 = clazz.getSuperclass().getInterfaces();
for(Class c : interfaces1){
    System.out.println(c);
}

/*
 * 获取运行时类所在的包
 */

@Test
public void test6(){
    Class clazz = Person.class;

    Package pack = clazz.getPackage();
    System.out.println(pack);
}

/*
 * 获取运行时类声明的注解
 */

@Test
public void test7(){
    Class clazz = Person.class;

    Annotation[] annotations = clazz.getAnnotations();
    for(Annotation annos : annotations){
        System.out.println(annos);
    }
}
```

反射应用三：调用运行时类的指定结构

调用指定的属性：

```

@Test
public void testField1() throws Exception {
    Class clazz = Person.class;

    //创建运行时类的对象
    Person p = (Person) clazz.newInstance();

    //1. getDeclaredField(String fieldName):获取运行时类中指定变量名的属性
    Field name = clazz.getDeclaredField("name");

    //2. 保证当前属性是可访问的
    name.setAccessible(true);
    //3. 获取、设置指定对象的此属性值
    name.set(p, "Tom");

    System.out.println(name.get(p));
}

```

调用指定的方法：

```

@Test
public void testMethod() throws Exception {

    Class clazz = Person.class;

    //创建运行时类的对象
    Person p = (Person) clazz.newInstance();

    /*
     * 1. 获取指定的某个方法
     *      getDeclaredMethod():参数1：指明获取的方法的名称 参数2：指明获取的方法的形参列表
     */
    Method show = clazz.getDeclaredMethod("show", String.class);
    //2. 保证当前方法是可访问的
    show.setAccessible(true);

    /*
     * 3. 调用方法的invoke():参数1：方法的调用者 参数2：给方法形参赋值的实参
     *      invoke()的返回值即为对应类中调用的方法的返回值。
     */
    Object returnValue = show.invoke(p, "CHN"); //String nation =
    p.show("CHN");
    System.out.println(returnValue);

    System.out.println("*****如何调用静态方法*****");
    System.out.println("*****");

    // private static void showDesc()

    Method showDesc = clazz.getDeclaredMethod("showDesc");
    showDesc.setAccessible(true);
    //如果调用的运行时类中的方法没返回值，则此invoke()返回null
    // Object retVal = showDesc.invoke(null);
    Object retVal = showDesc.invoke(Person.class);
    System.out.println(retVal); //null
}

}

```

调用指定的构造器：

```
@Test  
public void testConstructor() throws Exception {  
    Class clazz = Person.class;  
  
    //private Person(String name)  
    /*  
     * 1. 获取指定的构造器  
     * getDeclaredConstructor():参数：指明构造器的参数列表  
     */  
  
    Constructor constructor = clazz.getDeclaredConstructor(String.class);  
  
    //2. 保证此构造器是可访问的  
    constructor.setAccessible(true);  
  
    //3. 调用此构造器创建运行时类的对象  
    Person per = (Person) constructor.newInstance("Tom");  
    System.out.println(per);  
}
```

1. 代理模式的原理：

使用一个代理将对象包装起来，然后用该代理对象取代原始对象。任何对原始对象的调用都要通过代理。代理对象决定是否以及何时将方法调用转到原始对象上。

2. 静态代理

2.1 举例：

实现Runnable接口的方法创建多线程。

```
Class MyThread implements Runnable{} //相当于被代理类
```

```
Class Thread implements Runnable{} //相当于代理类
```

```
main(){
```

```
    MyThread t = new MyThread();
```

```
    Thread thread = new Thread(t);
```

```
    thread.start(); //启动线程； 调用线程的run()
```

```
}
```

2.2 静态代理的缺点：

- ① 代理类和目标对象的类都是在编译期间确定下来，不利于程序的扩展。
- ② 每一个代理类只能为一个接口服务，这样一来程序开发中必然产生过多的代理。

3. 动态代理的特点：

动态代理是指客户通过代理类来调用其它对象的方法，并且是在程序运行时根据需要动态创建目标类的代理对象。

4. 动态代理的实现

4.1 需要解决的两个主要问题：

问题一：如何根据加载到内存中的被代理类，动态的创建一个代理类及其对象。
(通过Proxy.newProxyInstance()实现)

问题二：当通过代理类的对象调用方法a时，如何动态的去调用被代理类中的同名方法a。
(通过InvocationHandler接口的实现类及其方法invoke())

4.2 代码实现：

```
/*
 * 动态代理的举例
 *
 * @author shkstart
 * @create 2019 上午 10:18
 */

interface Human{
    String getBelief();
    void eat(String food);
}

//被代理类
class SuperMan implements Human{
```

```

@Override
public String getBelief() {
    return "I believe I can fly!";
}

@Override
public void eat(String food) {
    System.out.println("我喜欢吃" + food);
}

class HumanUtil{

    public void method1(){
        System.out.println("=====通用方法一=====");
    }

    public void method2(){
        System.out.println("=====通用方法二=====");
    }
}

class ProxyFactory{
    //调用此方法，返回一个代理类的对象。解决问题一
    public static Object getProxyInstance(Object obj){//obj:被代理类的对象
        MyInvocationHandler handler = new MyInvocationHandler();

        handler.bind(obj);

        return Proxy.newProxyInstance(obj.getClass().getClassLoader(),obj.getClass()
        .getInterfaces(),handler);
    }
}

class MyInvocationHandler implements InvocationHandler{

    private Object obj;//需要使用被代理类的对象进行赋值

    public void bind(Object obj){
        this.obj = obj;
    }

    //当我们通过代理类的对象，调用方法a时，就会自动的调用如下的方法：invoke()
    //将被代理类要执行的方法a的功能就声明在invoke()中
    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {

        HumanUtil util = new HumanUtil();
        util.method1();

        //method: 即为代理类对象调用的方法，此方法也就作为被代理类对象要调用的方法
        //obj: 被代理类的对象
        Object returnValue = method.invoke(obj,args);

        util.method2();

        //上述方法的返回值就作为当前类中的invoke()的返回值。
        return returnValue;
    }
}

public class ProxyTest {

    public static void main(String[] args) {
        SuperMan superMan = new SuperMan();
        //proxyInstance:代理类的对象
        Human proxyInstance = (Human) ProxyFactory.getProxyInstance(superMan);
        //当通过代理类对象调用方法时，会自动的调用被代理类中同名的方法
        String belief = proxyInstance.getBelief();
        System.out.println(belief);
        proxyInstance.eat("四川麻辣烫");
    }
}

```

```
System.out.println("*****");
NikeClothFactory nikeClothFactory = new NikeClothFactory();
ClothFactory proxyClothFactory = (ClothFactory) ProxyFactory.getProxyInstance(nikeClothFactory);
proxyClothFactory.produceCloth();
}
```

体会：反射的动态性。



1. Lambda表达式使用前后的对比:

举例一:

```
@Test
public void test1(){

    Runnable r1 = new Runnable() {
        @Override
        public void run() {
            System.out.println("我爱北京天安门");
        }
    };

    r1.run();

    System.out.println("*****");

    Runnable r2 = () -> System.out.println("我爱北京故宫");

    r2.run();
}
```

举例二:

```
@Test
public void test2(){

    Comparator<Integer> com1 = new Comparator<Integer>() {
        @Override
        public int compare(Integer o1, Integer o2) {
            return Integer.compare(o1,o2);
        }
    };

    int compare1 = com1.compare(12,21);
    System.out.println(compare1);

    System.out.println("*****");
    //Lambda表达式的写法
    Comparator<Integer> com2 = (o1,o2) -> Integer.compare(o1,o2);

    int compare2 = com2.compare(32,21);
    System.out.println(compare2);

    System.out.println("*****");
    //方法引用
    Comparator<Integer> com3 = Integer :: compare;

    int compare3 = com3.compare(32,21);
    System.out.println(compare3);
}
```

2. Lambda表达式的基本语法:

- * 1. 举例: $(o1, o2) \rightarrow \text{Integer.compare}(o1, o2)$;
- * 2. 格式:
 - * \rightarrow :Lambda操作符 或 箭头操作符
 - * \rightarrow 左边: Lambda形参列表 (其实就是接口中的抽象方法的形参列表)

* -> 右边: *Lambda*体 (其实就是重写的抽象方法的方法体)

3. 如何使用: 分为六种情况

语法格式一: 无参, 无返回值

```
Runnable r1 = () -> {System.out.println("Hello Lambda!");};
```

语法格式二: *Lambda* 需要一个参数, 但是没有返回值。

```
Consumer<String> con = (String str) -> {System.out.println(str);};
```

语法格式三: 数据类型可以省略, 因为可由编译器推断得出, 称为“类型推断”

```
Consumer<String> con = (str) -> {System.out.println(str);};
```

语法格式四: *Lambda* 若只需要一个参数时, 参数的小括号可以省略

```
Consumer<String> con = str -> {System.out.println(str);};
```

语法格式五: *Lambda* 需要两个或以上的参数, 多条执行语句, 并且可以有返回值

```
Comparator<Integer> com = (x,y) -> {
    System.out.println("实现函数式接口方法!");
    return Integer.compare(x,y);
};
```

语法格式六: 当 *Lambda* 体只有一条语句时, *return* 与大括号若有, 都可以省略

```
Comparator<Integer> com = (x,y) -> Integer.compare(x, y);
```

总结六种情况:

-> 左边: *Lambda*形参列表的参数类型可以省略(类型推断); 如果*Lambda*形参列表只一个参数, 其一对()也可以省略

-> 右边: *Lambda*体应该使用一对{}包裹; 如果*Lambda*体只一条执行语句(可能是*return*语句, 省略这一对{}和*return*关键字)

1. 函数式接口的使用说明

- > 如果一个接口中，只声明了一个抽象方法，则此接口就称为函数式接口。
- > 我们可以在一个接口上使用 `@FunctionalInterface` 注解，这样做可以检查它是否是一个函数式接口。
- > `Lambda`表达式的本质：作为函数式接口的实例

2. Java8中关于Lambda表达式提供的4个基本的函数式接口：

具体使用：

函数式接口	参数类型	返回类型	用途
Consumer<T> 消费型接口	T	void	对类型为T的对象应用操作，包含方法： <code>void accept(T t)</code>
Supplier<T> 供给型接口	无	T	返回类型为T的对象，包含方法： <code>T get()</code>
Function<T, R> 函数型接口	T	R	对类型为T的对象应用操作，并返回结果。结果是R类型的对象。包含方法： <code>R apply(T t)</code>
Predicate<T> 断定型接口	T	boolean	确定类型为T的对象是否满足某约束，并返回 boolean 值。包含方法： <code>boolean test(T t)</code>

3. 总结

3.1 何时使用lambda表达式？

当需要对一个函数式接口实例化的时候，可以使用lambda表达式。

3.2 何时使用给定的函数式接口？

如果我们开发中需要定义一个函数式接口，首先看看在已有的jdk提供的函数式接口是否提供了

能满足需求的函数式接口。如果有，则直接调用即可，不需要自己再自定义了。

方法引用

方法引用

1. 理解：

方法引用可以看做是Lambda表达式深层次的表达。换句话说，方法引用就是Lambda表达式，也就是函数式接口的一个实例，通过方法的名字来指向一个方法。

2. 使用情境：

当要传递给Lambda体的操作，已经实现的方法了，可以使用方法引用！

3. 格式：

类(或对象) :: 方法名

4. 分为如下的三种情况：

- * 情况1 对象 :: 非静态方法
- * 情况2 类 :: 静态方法
- *
- * 情况3 类 :: 非静态方法

5. 要求：

- > 要求接口中的抽象方法的形参列表和返回值类型与方法引用的方法的形参列表和返回值类型相同！（针对于情况1和情况2）
- > 当函数式接口方法的第一个参数是需要引用方法的调用者，并且第二个参数是需要引用方法的参数(或无参数)时：ClassName::methodName（针对于情况3）

6. 使用建议：

如果给函数式接口提供实例，恰好满足方法引用的使用情境，大家就可以考虑使用方法引用给函数式接口提供实例。如果大家不熟悉方法引用，那么还可以使用Lambda表达式。

7. 使用举例：

```
// 情况一：对象 :: 实例方法
//Consumer中的void accept(T t)
//PrintStream中的void println(T t)
@Test
public void test1() {
    Consumer<String> con1 = str -> System.out.println(str);
    con1.accept("北京");

    System.out.println("*****");
    PrintStream ps = System.out;
    Consumer<String> con2 = ps::println;
    con2.accept("beijing");
}

//Supplier中的T get()
//Employee中的String getName()
@Test
public void test2() {
    Employee emp = new Employee(1001, "Tom", 23, 5600);

    Supplier<String> sup1 = () -> emp.getName();
    System.out.println(sup1.get());

    System.out.println("*****");
```

```

Supplier<String> sup2 = emp::getName;
System.out.println(sup2.get());

}

// 情况二: 类 :: 静态方法
//Comparator中的int compare(T t1,T t2)
//Integer中的int compare(T t1,T t2)
@Test
public void test3() {
    Comparator<Integer> com1 = (t1,t2) -> Integer.compare(t1,t2);
    System.out.println(com1.compare(12,21));

    System.out.println("*****");
}

Comparator<Integer> com2 = Integer::compare;
System.out.println(com2.compare(12,3));

}

//Function中的R apply(T t)
//Math中的Long round(Double d)
@Test
public void test4() {
    Function<Double,Long> func = new Function<Double, Long>() {
        @Override
        public Long apply(Double d) {
            return Math.round(d);
        }
    };

    System.out.println("*****");

    Function<Double,Long> func1 = d -> Math.round(d);
    System.out.println(func1.apply(12.3));

    System.out.println("*****");

    Function<Double,Long> func2 = Math::round;
    System.out.println(func2.apply(12.6));
}

// 情况: 类 :: 实例方法 (难度)
// Comparator中的int compare(T t1,T t2)
// String中的int t1.compareTo(t2)
@Test
public void test5() {
    Comparator<String> com1 = (s1,s2) -> s1.compareTo(s2);
    System.out.println(com1.compare("abc","abd"));

    System.out.println("*****");

    Comparator<String> com2 = String :: compareTo;
    System.out.println(com2.compare("abd","abm"));
}

//BiPredicate中的boolean test(T t1, T t2);
//String中的boolean t1.equals(t2)
@Test
public void test6() {
    BiPredicate<String,String> pre1 = (s1,s2) -> s1.equals(s2);
    System.out.println(pre1.test("abc","abc"));
}

```

```
System.out.println("*****");
BiPredicate<String, String> pre2 = String :: equals;
System.out.println(pre2.test("abc", "abd"));
}

// Function 中的 R apply(T t)
// Employee 中的 String getName();
@Test
public void test7() {
    Employee employee = new Employee(1001, "Jerry", 23, 6000);

    Function<Employee, String> func1 = e -> e.getName();
    System.out.println(func1.apply(employee));

    System.out.println("*****");

    Function<Employee, String> func2 = Employee::getName;
    System.out.println(func2.apply(employee));
}

}
```

构造器引用与数组引用

1. 构造器引用格式:

类名::new

2. 构造器引用使用要求:

和方法引用类似，函数式接口的抽象方法的形参列表和构造器的形参列表一致。抽象方法的返回值类型即为构造器所属的类的类型

3. 构造器引用举例:

```
//Supplier中的T get()
//Employee的空参构造器: Employee()
@Test
public void test1(){

    Supplier<Employee> sup = new Supplier<Employee>() {
        @Override
        public Employee get() {
            return new Employee();
        }
    };
    System.out.println("*****");

    Supplier<Employee> sup1 = () -> new Employee();
    System.out.println(sup1.get());

    System.out.println("*****");

    Supplier<Employee> sup2 = Employee :: new;
    System.out.println(sup2.get());
}

//Function中的R apply(T t)
@Test
public void test2(){
    Function<Integer,Employee> func1 = id -> new Employee(id);
    Employee employee = func1.apply(1001);
    System.out.println(employee);

    System.out.println("*****");

    Function<Integer,Employee> func2 = Employee :: new;
    Employee employee1 = func2.apply(1002);
    System.out.println(employee1);

}

//BiFunction中的R apply(T t,U u)
@Test
public void test3(){
    BiFunction<Integer,String,Employee> func1 = (id,name) -> new Employee
(id,name);
    System.out.println(func1.apply(1001,"Tom"));

    System.out.println("*****");

    BiFunction<Integer,String,Employee> func2 = Employee :: new;
    System.out.println(func2.apply(1002,"Tom"));

}
```

4. 数组引用格式:

数组类型[] :: new

5. 数组引用举例:

```
//Function 中的 R apply(T t)
@Test
public void test4(){
    Function<Integer, String[]> func1 = length -> new String[length];
    String[] arr1 = func1.apply(5);
    System.out.println(Arrays.toString(arr1));

    System.out.println("*****");
    Function<Integer, String[]> func2 = String[] :: new;
    String[] arr2 = func2.apply(10);
    System.out.println(Arrays.toString(arr2));
}
```

1. Stream API的理解：

1.1 Stream关注的是对数据的运算，与CPU打交道
集合关注的是数据的存储，与内存打交道

1.2 java8提供了一套api，使用这套api可以对内存中的数据进行过滤、排序、映射、归约等操作。类似于sql对数据库中表的相关操作。

2. 注意点：

- * ① Stream自己不会存储元素。
- * ② Stream不会改变源对象。相反，他们会返回一个持有结果的新Stream。
- * ③ Stream操作是延迟执行的。这意味着他们会等到需要结果的时候才执行。

3. Stream的使用流程：

- * ① Stream的实例化
- * ② 一系列的中间操作（过滤、映射、...）
- * ③ 终止操作

4. 使用流程的注意点：

- * 4.1 一个中间操作链，对数据源的数据进行处理
- * 4.2 一旦执行终止操作，就执行中间操作链，并产生结果。之后，不会再被使用

5. 步骤一： Stream实例化

```
// 创建 Stream 方式一：通过集合
@Test
public void test1(){
    List<Employee> employees = EmployeeData.getEmployees();

    //      default Stream<E> stream() : 返回一个顺序流
    Stream<Employee> stream = employees.stream();

    //      default Stream<E> parallelStream() : 返回一个并行流
    Stream<Employee> parallelStream = employees.parallelStream();

}

// 创建 Stream 方式二：通过数组
@Test
public void test2(){
    int[] arr = new int[]{1,2,3,4,5,6};
    // 调用 Arrays 类的 static <T> Stream<T> stream(T[] array) : 返回一个流
    IntStream stream = Arrays.stream(arr);

    Employee e1 = new Employee(1001, "Tom");
    Employee e2 = new Employee(1002, "Jerry");
    Employee[] arr1 = new Employee[]{e1,e2};
    Stream<Employee> stream1 = Arrays.stream(arr1);

}

// 创建 Stream 方式三：通过 Stream 的 of()
@Test
public void test3(){

    Stream<Integer> stream = Stream.of(1, 2, 3, 4, 5, 6);
}
```

```

}

// 创建 Stream 方式四：创建无限流
@Test
public void test4(){

    // 迭代
    // public static<T> Stream<T> iterate(final T seed, final
    UnaryOperator<T> f)
        // 遍历前10个偶数
        Stream.iterate(0, t -> t + 2).limit(10).forEach
    (System.out::println);

    // 生成
    // public static<T> Stream<T> generate(Supplier<T> s)
        Stream.generate(Math::random).limit(10).forEach
    (System.out::println);

}

```

6. 步骤二：中间操作

1-筛选与切片

方法	描述
filter(Predicate p)	接收 Lambda，从流中排除某些元素
distinct()	筛选，通过流所生成元素的 hashCode() 和 equals() 去除重复元素
limit(long maxSize)	截断流，使其元素不超过给定数量
skip(long n)	跳过元素，返回一个扔掉了前 n 个元素的流。若流中元素不足 n 个，则返回一个空流。与 limit(n) 互补

2-映射

方法	描述
map(Function f)	接收一个函数作为参数，该函数会被应用到每个元素上，并将其映射成一个新的元素。
mapToDouble(ToDoubleFunction f)	接收一个函数作为参数，该函数会被应用到每个元素上，产生一个新的 DoubleStream。
mapToInt(ToIntFunction f)	接收一个函数作为参数，该函数会被应用到每个元素上，产生一个新的 IntStream。
mapToLong(ToLongFunction f)	接收一个函数作为参数，该函数会被应用到每个元素上，产生一个新的 LongStream。
flatMap(Function f)	接收一个函数作为参数，将流中的每个值都换成另一个流，然后把所有流连接成一个流

3-排序

方法	描述
sorted()	产生一个新流，其中按自然顺序排序
sorted(Comparator com)	产生一个新流，其中按比较器顺序排序

7. 步骤三：终止操作

1-匹配与查找

方法	描述
allMatch(Predicate p)	检查是否匹配所有元素
anyMatch(Predicate p)	检查是否至少匹配一个元素
noneMatch(Predicate p)	检查是否没有匹配所有元素
findFirst()	返回第一个元素
findAny()	返回当前流中的任意元素

方法	描述
count()	返回流中元素总数
max(Comparator c)	返回流中最大值
min(Comparator c)	返回流中最小值
forEach(Consumer c)	内部迭代(使用 Collection 接口需要用户去做迭代, 称为外部迭代。相反, Stream API 使用内部迭代——它帮你把迭代做了)

2-归约

方法	描述
reduce(T iden, BinaryOperator b)	可以将流中元素反复结合起来，得到一个值。返回 T
reduce(BinaryOperator b)	可以将流中元素反复结合起来，得到一个值。返回 Optional<T>

备注：map 和 reduce 的连接通常称为 map-reduce 模式，因 Google 用它来进行网络搜索而出名。

3-收集

方法	描述
collect(Collector c)	将流转换为其他形式。接收一个 Collector 接口的实现，用于给 Stream 中元素做汇总的方法

Collector 需要使用 Collectors 提供实例。

16.4 强大的Stream API: Collectors



方法	返回类型	作用
toList	List<T>	把流中元素收集到List
<code>List<Employee> emps= list.stream().collect(Collectors.toList());</code>		
toSet	Set<T>	把流中元素收集到Set
<code>Set<Employee> emps= list.stream().collect(Collectors.toSet());</code>		
toCollection	Collection<T>	把流中元素收集到创建的集合
<code>Collection<Employee> emps =list.stream().collect(Collectors.toCollection(ArrayList::new));</code>		

java.util.Optional类

1. 理解：为了解决java中的空指针问题而生！

Optional<T> 类(java.util.Optional) 是一个容器类，它可以保存类型T的值，代表这个值存在。或者仅仅保存null，表示这个值不存在。原来用 null 表示一个值不存在，现在 Optional 可以更好的表达这个概念。并且可以避免空指针异常。

2. 常用方法：

```

@Test
public void test1(){
    //empty(): 创建的Optional对象内部的value = null
    Optional<Object> op1 = Optional.empty();
    if(!op1.isPresent()){//Optional封装的数据是否包含数据
        System.out.println("数据为空");
    }
    System.out.println(op1);
    System.out.println(op1.isPresent());
    //如果Optional封装的数据value为空，则get()报错。否则，value不为空时，返回value。
    //      System.out.println(op1.get());
}

@Test
public void test2(){
    String str = "hello";
    //str = null;
    //of(T t): 封装数据t生成Optional对象。要求t非空，否则报错。
    Optional<String> op1 = Optional.of(str);
    //get()通常与of()方法搭配使用。用于获取内部的封装的数据value
    String str1 = op1.get();
    System.out.println(str1);

}

@Test
public void test3(){
    String str = "beijing";
    str = null;
    //ofNullable(T t) : 封装数据t赋给Optional内部的value。不要求t非空
    Optional<String> op1 = Optional.ofNullable(str);
    //orElse(T t1): 如果Optional内部的value非空，则返回此value值。如果value为空，则返回t1。
    String str2 = op1.orElse("shanghai");

    System.out.println(str2);//

}

```

3. 典型练习：

能保证如下的方法执行中不会出现空指针的异常。

```
//使用Optional类的getGirlName():
public String getGirlName2(Boy boy){

    Optional<Boy> boyOptional = Optional.ofNullable(boy);
    //此时的boy1一定非空
    Boy boy1 = boyOptional.orElse(new Boy(new Girl("迪丽热巴")));

    Girl girl = boy1.getGirl();

    Optional<Girl> girlOptional = Optional.ofNullable(girl);
    //girl1一定非空
    Girl girl1 = girlOptional.orElse(new Girl("古力娜扎"));

    return girl1.getName();
}

@Test
public void test5(){
    Boy boy = null;
    boy = new Boy();
    boy = new Boy(new Girl("苍老师"));
    String girlName = getGirlName2(boy);
    System.out.println(girlName);

}
```

Java 9新特性

具体特性可直接查看第17章ppt即可。由于已经比较清晰了，这里不再单独复习记录了。

祝大家学业有成！

Java 10新特性

具体特性可直接查看第17章ppt即可。由于已经比较清晰了，这里不再单独复习记录了。

祝大家学业有成！

Java 11新特性

具体特性可直接查看第17章ppt即可。由于已经比较清晰了，这里不再单独复习记录了。

祝大家学业有成！

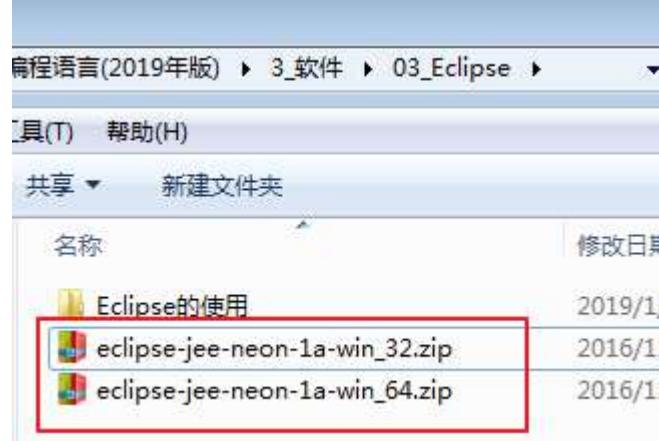
Eclipse的配置

1. eclipse的安装:

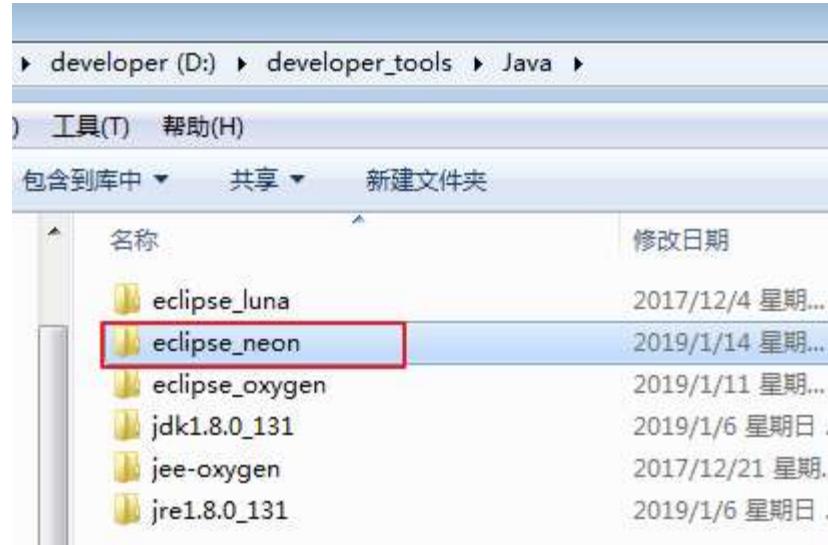
解压以后，把解压文件放在没有中文和空格的路径下，直接选择eclipse.exe执行。

2. 区分安装目录与workspace

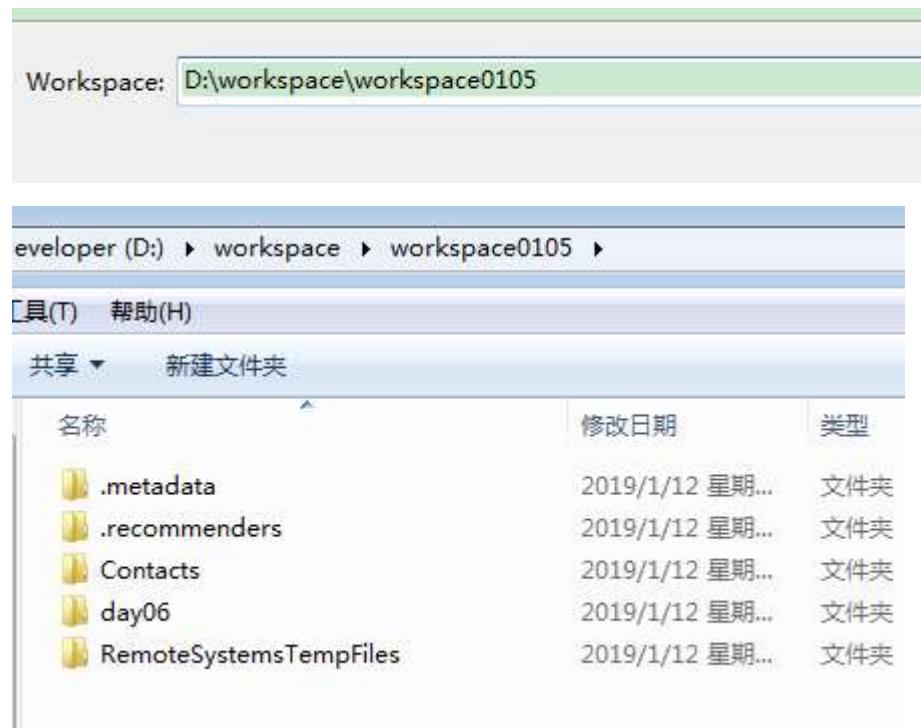
>软件的文件存放的位置：



>安装目录：



>workspace：代码的存放位置



3. 具体的配置，见《Eclipse的使用配置.doc》

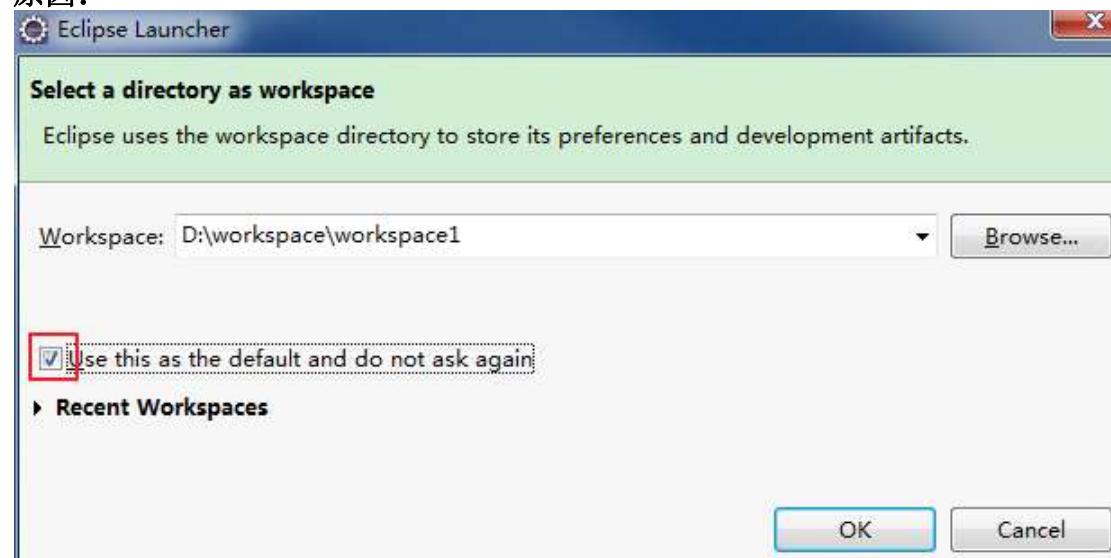
1. 双击Eclipse启动图标，不能正常启动Eclipse

启动不了的原因很多种，这里需要大家从如下几个方面排查：

1. 环境变量是否正确配置，需要在命令行输入javac.exe或java.exe进行检查
2. 是否正确的安装了JDK和JRE
3. 安装的JDK的版本（32位还是64位，必须与Eclipse版本一致）
4. 修改Eclipse安装目录下的eclipse.ini配置文件

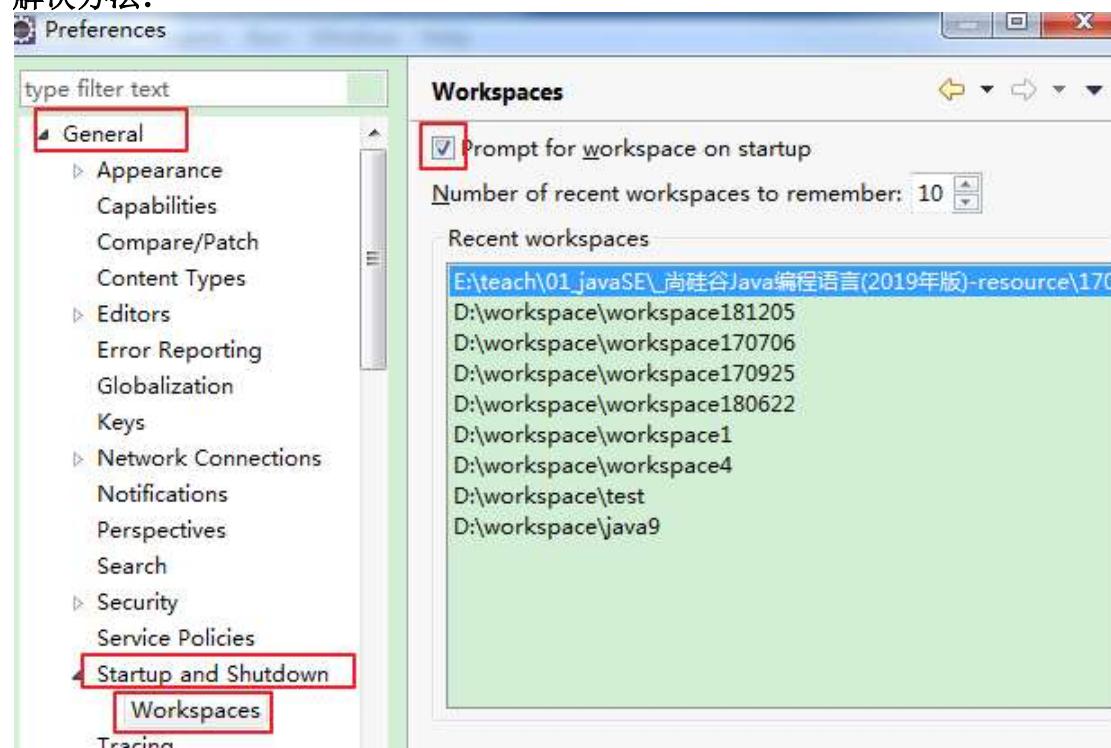
2. 进入Eclipse时，没可选的workspace

原因：



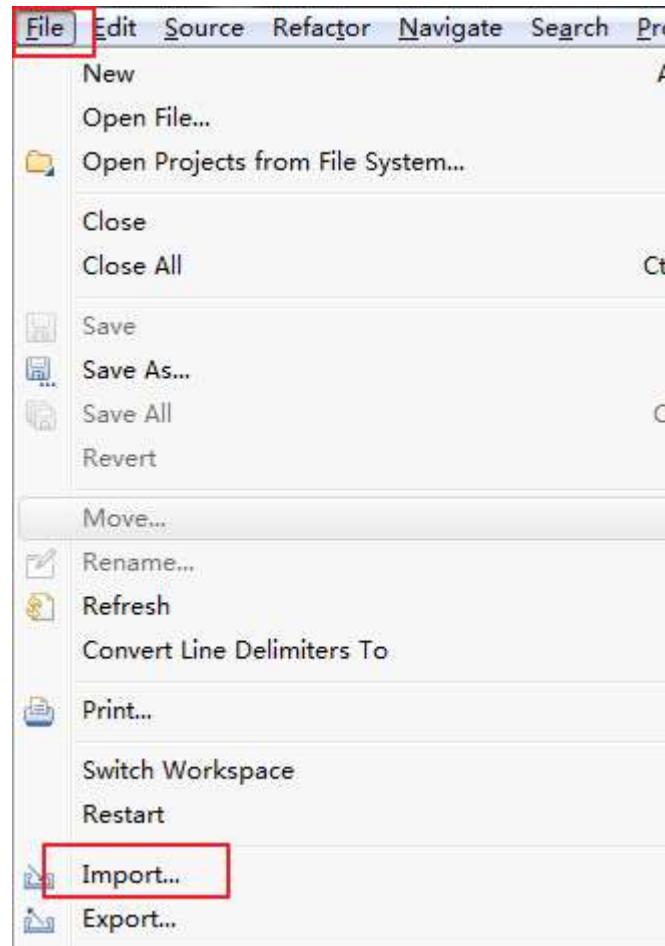
由于勾了上述红框，所以再次启动Eclipse时，不再显示可选的workspace。

解决办法：

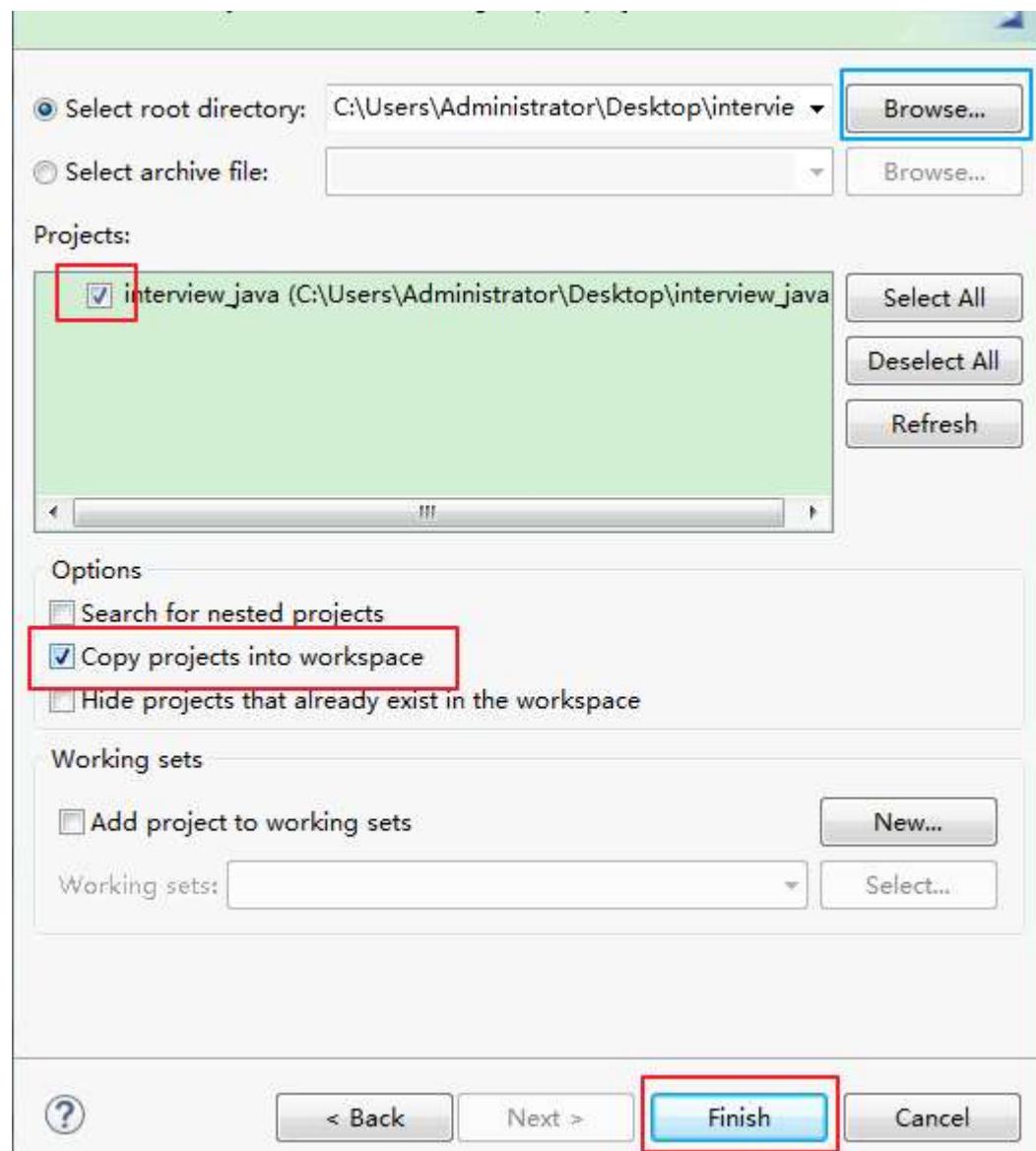
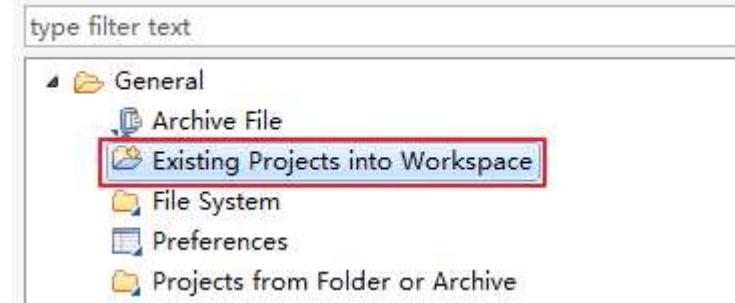


在windows – Preferences下，将上述的红框勾即可。

3. 如何导入已的工程



接着，



通过点击蓝框择要导入的Java工程，然后勾上述的两个红框，确认即可。

4. 如何导入已的一个源文件

直接复制（ctrl + c）此源文件，直接在指定的工程的包下粘贴（ctrl + v）即可。

5. 如何删除一个工程



6. 工程中的代码乱码怎么办

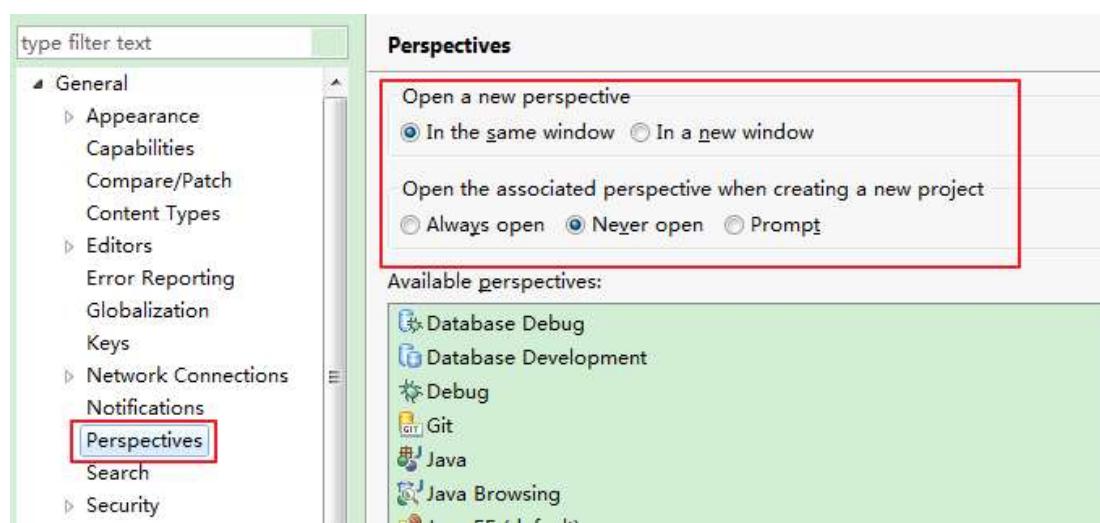
原因:

出现乱码的代码所使用的字符编码集与工程设置使用的字符编码集不一致导致的。

如何解决:

建议修改乱码文件的字符编码集即可。

7. 运行程序，误择了Java透视图，如何调整为JavaEE的



将设置修改为上图设置的情况即可解决。

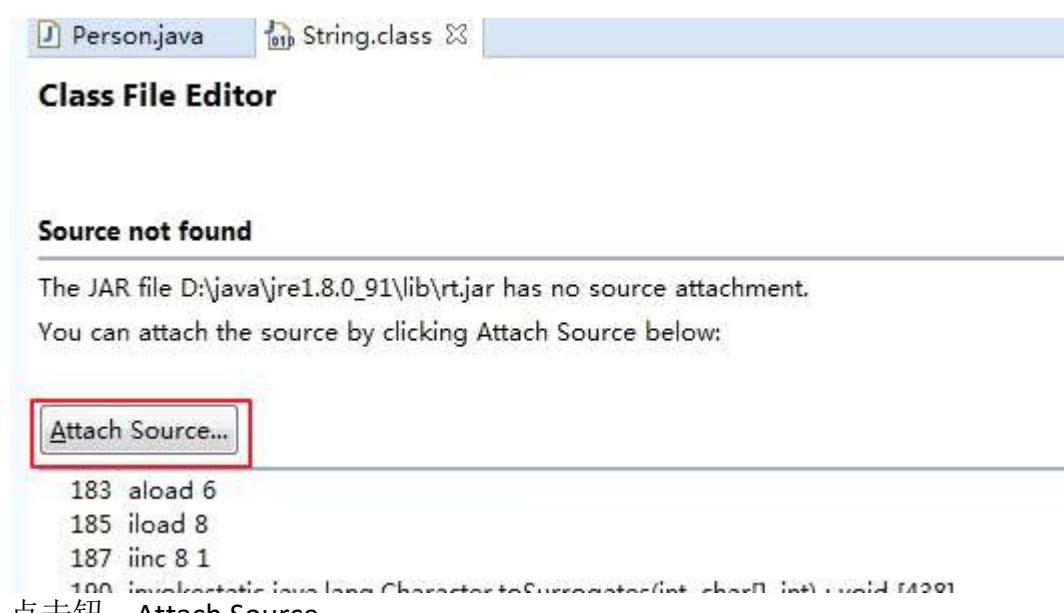
8. 再创建一个workspace，之前的设置怎么没了

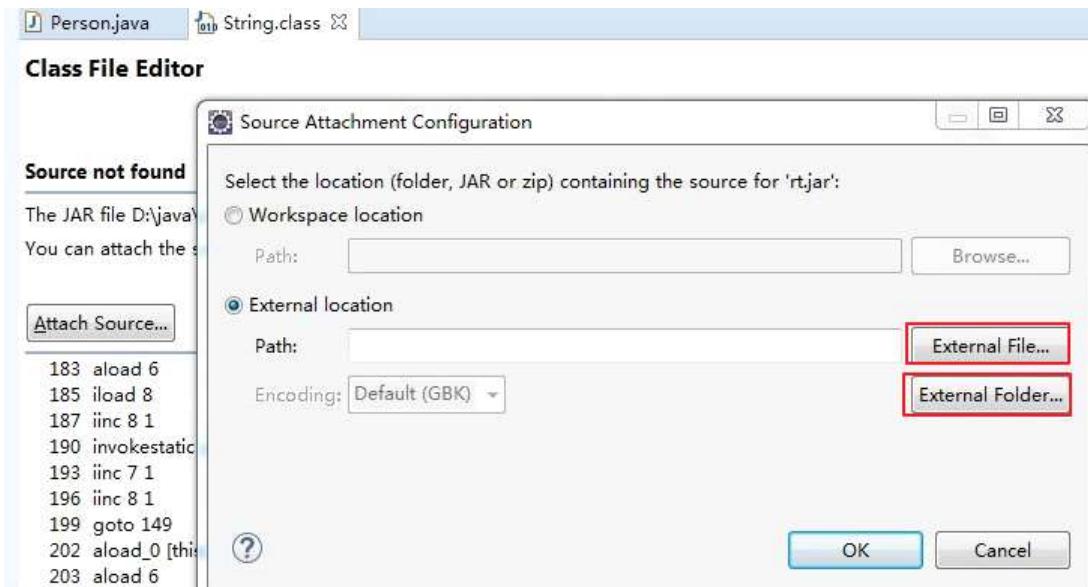
原因:

我们对Eclipse做的设置，只对当前使用的workspace有效。设置的数据保存在相应workspace文件夹下的.metadata文件夹中。如果创建了新的workspace，则需要重新设置。

9. 如何在Eclipse中查看Java类库源代码呢？

在代码中，综合使用ctrl + 鼠标左键点击指定结构时，可以调出如下界面：

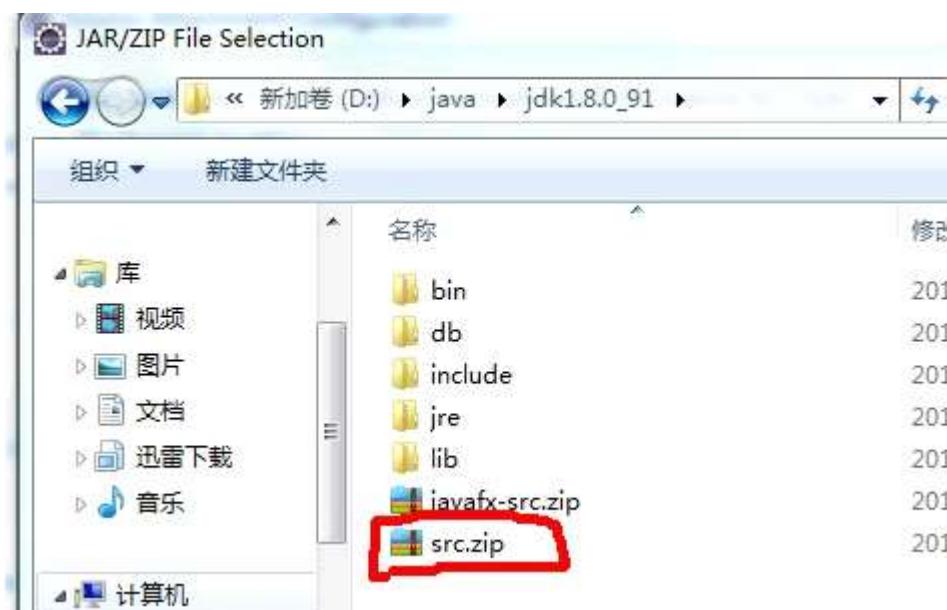




要导入的源码如果是文件方式存在，则择： External File

要导入的源码如果是文件夹方式存在，则择： External Folder

这里jdk源码以src.zip方式存在，所以择： External File



中确认即可。

10. 如何在编写的代码中显示程序员的相关信息

依次择：

Window-->Preferences-->Java-->Code Style-->Code Templates

点击Comments

(1 找到Types 然后双击填入以下几个信息即可

```
/**  
 * @Description  
 * @author shkstart Email:shkstart@126.com  
 * @version  
 * @date ${date}${time}  
 */
```

框中红色的，大家填写自己的信息即可。

(2 找到Methods 然后双击填入以下几个信息即可

```
/**  
 * @Description  
 * @author shkstart  
 * @date ${date}${time}  
 * ${tags}  
 */
```

框中红色的，大家填写自己的信息即可。

常用快捷键*** Eclipse中的快捷键:**

- * 1. 补全代码的声明: `alt + /`
- * 2. 快速修复: `ctrl + 1`
- * 3. 使用单行注释: `ctrl + /`
- * 4. 使用多行注释: `ctrl + shift + /`
- * 5. 取消多行注释: `ctrl + shift + \`
- * 6. 复制指定行的代码: `ctrl + alt + down` 或 `ctrl + alt + up`
- * 7. 删除指定行的代码: `ctrl + d`
- * 8. 上下移动代码: `alt + up` 或 `alt + down`
- * 9. 切换到下一行代码空位: `shift + enter`
- * 10. 切换到上一行代码空位: `ctrl + shift + enter`
- * 11. 如何查看源码: `ctrl +` 选中指定的结构 或 `ctrl + shift + t`
- * 12. 退回到前一个编辑的页面: `alt + left`
- * 13. 进入到下一个编辑的页面(针对于上面那条来说的): `alt + right`
- * 14. 光标选中指定的类, 查看继承树结构: `ctrl + t`
- * 15. 复制代码: `ctrl + c`
- * 16. 撤销: `ctrl + z`
- * 17. 反撤销: `ctrl + y`
- * 18. 剪切: `ctrl + x`
- * 19. 粘贴: `ctrl + v`
- * 20. 保存: `ctrl + s`
- * 21. 全选: `ctrl + a`
- * 22. 格式化代码: `ctrl + shift + f`
- * 23. 选中数行, 整体往后移动: `tab`
- * 24. 选中数行, 整体往前移动: `shift + tab`
- * 25. 在当前类中, 显示类结构, 并支持搜索指定的方法、属性等: `ctrl + o`
- * 26. 批量修改指定的变量名、方法名、类名等: `alt + shift + r`
- * 27. 选中的结构的大小写的切换: 变成大写: `ctrl + shift + x`
- * 28. 选中的结构的大小写的切换: 变成小写: `ctrl + shift + y`
- * 29. 批量导包: `ctrl + shift + o`
- * 30. 调出生成getter/setter/构造器等结构: `alt + shift + s`
- * 31. 显示当前选择资源(工程 or 文件)的属性: `alt + enter`
- * 32. 快速查找: 参照选中的Word快速定位到下一个 : `ctrl + k`
- *
- * 33. 关闭当前窗口: `ctrl + w`
- * 34. 关闭所有的窗口: `ctrl + shift + w`

- * 35. 查看指定的结构使用过的地方: `ctrl + alt + g`
- * 36. 查找与替换: `ctrl + f`
- * 37. 最大化当前的View: `ctrl + m`
- * 38. 直接定位到当前行的首位: `home`
- * 39. 直接定位到当前行的末位: `end`

Debug调试

操作	作用
step into 跳入 (f5)	进入当前行所调用的方法中
step over 跳过 (f6)	执行完当前行的语句，进入下一行
step return 跳回 (f7)	执行完当前行所在的方法，进入下一行
drop to frame	回到当前行所在方法的第一行
resume 恢复	执行完当前行所在断点的所有代码，进入下一个断点，如果没有就结束
Terminate 终止	停止 JVM，后面的程序不会再执行

01 - IDEA 的介绍

IDEA，全称IntelliJ IDEA，是Java语言的集成开发环境，IDEA在业界被公认为是最好的java开发工具之一，尤其在智能代码助手、代码自动提示、重构、J2EE支持、Ant、JUnit、CVS整合、代码审查、创新的GUI设计等方面的功能可以说是超常的。

IntelliJ IDEA 在2015年的官网上这样介绍自己：

Excel at enterprise, mobile and web development with Java, Scala and Groovy, with all the latest modern technologies and frameworks available out of the box.

02 - IDEA的下载和安装

下载：官网下载

<https://www.jetbrains.com/idea/download/#section=windows>

IDEA分为两个版本：**旗舰版(Ultimate)**和**社区版(Community)**。

旗舰版收费(限30天免费试用)，社区版免费，这和Eclipse很大区别。

安装：傻瓜式安装

03 - 启动IDEA并完成HelloWorld的运行

04 - 常用配置的设置

05 - 快捷键的设置

06 - 模板的使用

详细使用，参考文档《尚硅谷_宋红康_IntelliJIDEA的安装、配置与使用.doc》

项目要求

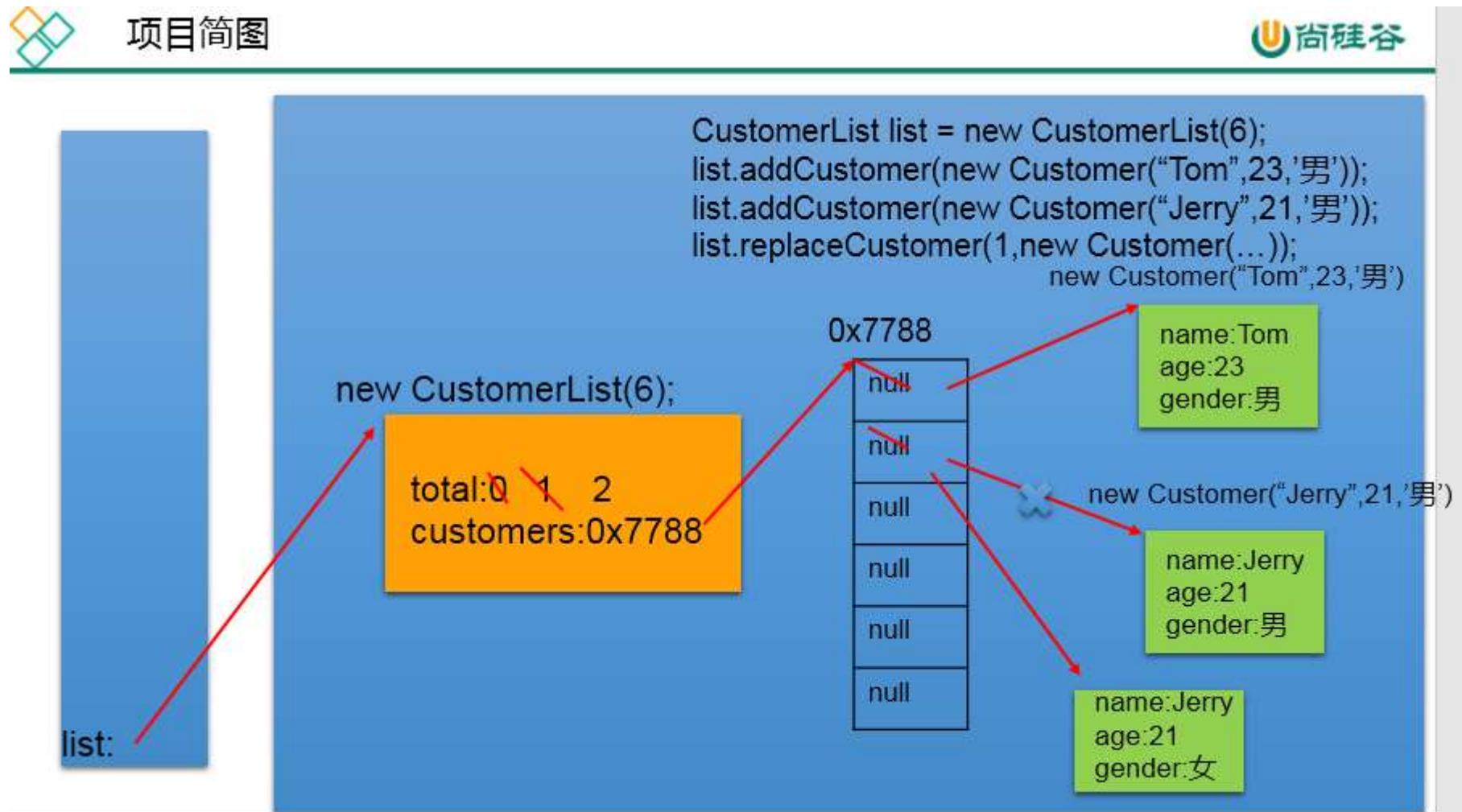
1. 至少独立完成一遍以上的项目代码
2. 积累完成项目的过程中常见的bug的调试

方式一：“硬”看，必要时，添加输出语句。

方式二：Debug

3. 将顺思路，强化逻辑

4. 对象、数组等内存结构的解析



注：项目二

5. 遵守编码的规范，标识符的命名规范等

```
int i1 = 10;
```

```
int total = 0;
```

6. 在类前，方法前，方法内具体逻辑的实现步骤等添加必要的注释。

类前、方法前、属性前：文档注释。

逻辑步骤：单行、多行注释。